UNIVERSITY OF TARTU
Faculty of Science and Technology
Institute of Computer Science
Cybersecurity Curriculum

Olivier Levasseur

Model-Driven Engineering of Blockchain Oracles

Master's Thesis (24 ECTS)

Supervisor(s): Mubashar Iqbal, MSc
Raimundas Matulevičius, PhD

Tartu 2022

Model-Driven Engineering of Blockchain Oracles

Abstract:

Blockchain is a decentralized ledger technology that provides data storage with strong integrity properties without the need for a trusted authority. Model-driven engineering is a software engineering discipline that aims at working with domain models instead of source code. Blockchain oracles are software components that can provide a communication channel between traditional off-chain applications and smart contracts. Model-driven engineering solutions have started being used to facilitate the development of blockchain-based applications through domain models. However, our literature review shows that there are limitations in the identified solutions to model blockchain oracles and that some types of oracles cannot be modeled through these solutions. In this thesis, we define a meta-model to illustrate the main concepts of the blockchain oracle domain and the relationships between them. We define a UML profile that extends the UML sequence diagram to include elements to model blockchain oracles. Our model also includes security components to provide encrypted communication with the oracles. A MagicDraw plugin is developed to implement transformation rules that automatically translate an oracle model into a standalone application. This results in a model-driven engineering solution that facilitates the modeling of secure blockchain oracles. The contribution of this thesis is evaluated based on functional and security requirements using an evaluation scenario that involves four different types of oracles interacting with an off-chain application.

Keywords: Blockchain, Blockchain oracles, Model-driven engineering

CERCS: T120 Systems engineering, computer technology

Plokiahela oraakli mudelipõhine projekteerimine

Lühikokkuvõte:

Plokiahel on tugevate terviklikkusomadustega andmetalletust pakkuv hajusraamatu tehnoloogia, mis ei vaja usaldatavat osapoolt. Mudelipõhine projekteerimine on tarkvaratehnika distsipliin, mis lähtekoodi asemel töötab domeenimudelitega. Plokiahela oraakliks nimetame tarkvarakomponenti, mis liidestab klassikalisi ahelaväliseid rakendusi plokiahelapõhiste lepingumonitoridega. Mudelipõhiseid lahendusi on hakatud kasutama, hõlbustamaks plokiahelapõhiste rakenduste arendamist domeenimudelite kaudu. Meie olemasolema kirjanduse läbivaatus näitab, et plokiahela oraaklite modelleerimiseks leitud lahendustel on piiranguid ja teatud tüüpi oraakleid ei saa nende lahenduste kaudu modelleerida. Selles lõputöös defineerime metamudeli, et illustreerida plokiahela oraakli valdkonna põhimõisteid ja nendevahelisi seoseid. Määratleme UML-profiili, mis laiendab UML-i järgnevusskeemi, et hõlmata elemente plokiahela oraaklite modelleerimiseks. Meie mudel sisaldab ka turvakomponente, kirjeldamaks krüpteeritud sidet oraaklitega. Arendasime MagicDraw pistikprogrammi rakendamaks teisendusreegleid, mis tõlgivad oraaklipõhise mudeli automaatselt klassikaliseks rakenduseks. Töö tulemuseks on mudelipõhine lahendus, mis hõlbustab turvaliste plokiahela oraaklite modelleerimist. Oma tööd oleme analüüsinud funktsionaalsete ja turvanõuete alusel, kasutades hindamisstsenaariumit, mis hõlmab nelja erinevat tüüpi oraaklit, mis suhtlevad ahelavälise rakendusega.

Võtmesõnad: Plokiahel, Plokiahela oraakel, Mudelipõhine projekteerimine

CERCS: T120 - Süsteemitehnoloogia, arvutitehnoloogia

Contents

1 Introduction

Blockchain is a decentralized distributed ledger technology that uses consensus and cryptography to protect the data from being tampered. This results in a system with high availability and integrity which does not require a trusted authority [IM19]. Blockchain platforms like Ethereum have built-in support for smart contracts. These contracts act as decentralized applications that are executed and verified by all the nodes contributing to the blockchain. One limitation of these contracts is that their execution must be deterministic in order to be verified by many different nodes [He20]. This means that smart contracts cannot make calls to access data outside the blockchain since each of these calls could return different data each time. As a result, there is no native way for the blockchain to fetch external data. To address this limitation, oracles have been designed. *Blockchain oracles* are software components that allow communication between off-chain applications and blockchain applications [LXSY20]. Model-driven engineering is a software methodology that aims at working with domain models instead of source code and that generates the source code from a model [HZD+20]. Model-driven engineering of blockchain oracles is the use of model-driven engineering to model and automatically generate the source code of blockchain oracles.

Blockchain technology brings new opportunities to organizations by allowing them to operate in a trustless environment. The tamperproof nature of the blockchain gives confidence that data stored on the blockchain will not be modified by malicious actors. Smart contracts can be written to build applications whose execution is guaranteed to conform to the instructions of the immutable code. No third party is needed to ensure that each party behaves according to the contract and the ledger keeps a trace of each party's transaction which is useful for monitoring the state of the process, auditing purposes as well as coordination between different parties [CMO18, XLL+19, FN16].

However, writing blockchain applications requires blockchain-specific knowledge and is error-prone [FHB+19, GD19]. For instance, Ethereum transactions have a cost and without providing the right amount of transaction fees, the transaction is likely to never be added to the blockchain. As another example, because of the nature of the mechanism used to add transactions to the blockchain, parties making transactions on public blockchains should wait until a certain number of blocks has been added to the blockchain before assuming that the transaction is final. On Ethereum, the recommended amount of blocks is 11 [Fou] while the recommended number is 6 on Bitcoin [Wik]. As a final example, the interaction with most blockchain applications cannot be done through regular HTTP requests and must be done through blockchain nodes, which are different from one blockchain to another [TB20]. These examples show that blockchain applications have unique characteristics that have to be taken into consideration when designing and developing them and that not knowing these characteristics can result in unexpected behavior and potential security threats.

Model-driven engineering has already started being used to abstract the complexities

of blockchain application development [FHB+19, GD19]. Security features can also be integrated into the model making security an element of the design of the application. For instance, Blockchain Studio integrates access control policies into a BPMN model [MBH18]. As another example, FSolidM offers an access control extension, protection against reentrancy, and state unpredictability which are vulnerabilities of Ethereum smart contracts [ML18]. The code generation part of model-driven engineering (MDE) can be used to integrate best practices in the resulting source code of the application and avoid the most common development mistakes by using common design patterns [ML18]. However, our SLR shows that there are limitations in the current state of the art for modeling blockchain oracles, which are important elements of blockchain applications since, without oracles, smart contracts cannot request data that is not stored on the blockchain, like real-time data, exchange rates, or weather conditions [PXB+19].

In this thesis, a UML profile is designed to extend the standard UML language to include elements that are specific to the blockchain oracle domain. This UML profile contains UML stereotypes and tagged values that allow the user to model 4 different types of blockchain oracles, which will be described later. Then, a plugin is developed for the MagicDraw modeling tool. This plugin provides the functionality to verify the model and generate the oracle's source code from the model. The verification phase of the model ensures that all the required elements of the oracle are present, but also that the required security elements are present in the model. These security elements are used to generate the source code of an oracle that provides an authenticated HTTPS connection with its clients. The repository at `https://gitlab.com/olevasseur/mde-blockchain-oracles` contains an example of an oracle model, the UML profile, and the source code of the plugin developed in this thesis.

This thesis will attempt to answer one main research question: **[MRQ] How can model-driven engineering facilitate the development of secure blockchain oracles?** To answer this question, we divide it into 4 sub-research questions, corresponding to the literature review, requirements and design, implementation, and evaluation sections respectively. For the literature review, the following research question will be answered: **[RQ1] What are the strengths and weaknesses of the model-driven engineering solutions for blockchain-based applications?** In the requirements and design section, the following research question will be answered: **[RQ2] What are the requirements for a model-driven engineering solution for blockchain oracles?** In the contribution section, the following research question will be answered: **[RQ3] How to model secure blockchain oracles using model-driven engineering?** In the evaluation section, the following research question will be answered: **[RQ4] What is the usability of the proposed solution for MDE of blockchain oracles?**

The rest of the thesis is organized as follows. Section 2 presents the literature review that aims at identifying the strengths and weaknesses of the current state of the art of model-driven engineering of blockchain-based applications. Section 3 describes the

goals, use cases, functional requirements, and security requirements that the system aims to achieve. Section 4 presents the main aspects of the contribution of this thesis, namely the suggested model, the transformation rules between the model and the generated software, and an overview of the architecture of the overall system. Section 5 presents an evaluation scenario for validating the requirements described in Section 3. Section 6 concludes the thesis.

2 Literature Review

In this section, we use the systematic literature review (SLR) to find existing model-driven engineering solutions for blockchain-based applications. The objective of this SLR is to identify existing blockchain application modeling solutions and to extract and summarize the main blockchain constructs represented in the existing modeling solutions. The focus is put on smart contracts and oracles and the considered modeling solutions are the ones that are visual and/or intuitive even to non-blockchain experts i.e text-based solutions are excluded. At the time of writing this paper, no existing systematic literature was found on this specific topic. The literature review is done from a model-driven engineering perspective, meaning that the approaches that are suitable for automatic code generation are considered. In other words, solutions that would model blockchain applications to evaluate their latency or their resource consumption would be excluded unless the method is also a good candidate for code generation.

In this section, we are answering research question 1: **[RQ1] What are the strengths and weaknesses of the model-driven engineering solutions for blockchain-based applications?** To better answer this research question, we have divided it into three sub-research questions. The first sub-research question is **[RQ1.1] What are the main characteristics of existing model-driven engineering solutions?** This sub-research question aims at identifying general characteristics of MDE solutions such as the modeling language and the level of supported code generation. The second research question is **[RQ1.2] What are the main elements of blockchain applications that are found in existing modeling solutions?** This sub-research question aims at identifying which elements of the blockchain applications are represented in the modeling solutions, such as smart contracts, and transactions and business rules. The third sub-research question is **[RQ1.3] What types of oracles are modeled and how are they modeled?** This sub-research question aims at evaluating to which degree existing MDE solutions support blockchain oracles.

2.1 Literature Sources

The initial search for relevant papers was done through the ACM digital library, the IEEE digital library, ScienceDirect, and Scopus. Additional relevant papers were identified from the related work sections and citations of the papers identified in the initial search. Grey literature was also considered, but the relevant results that matched the inclusion criteria were already covered in academic or scientific papers.

2.2 Search Terms

The search terms were the following: *("blockchain" AND ("modeling" OR "modelling" OR "model-driven" OR "model-driven engineering"))*. The search terms *modeling* and

modelling are synonyms and can be used interchangeably, but papers usually use one or the other. Other search terms were considered and ended up being excluded. For example, *decentralized application* was considered, but initial results showed that this included papers that were not relevant, and all the relevant ones also seemed to contain the word *blockchain*. The same was true for the term *smart contract*. Papers were initially selected if these search terms were present in the title or the abstract of the paper.

2.3 Inclusion and Exclusion Criteria

The inclusion criteria were the following:

- Papers related to blockchain application modeling
- Papers related to model-driven engineering of blockchain applications

The exclusion criteria were the following:

- Papers shorter than 5 pages and not written in English
- Papers published before 2008 and not accessible freely
- Papers that are not modeling applications, e.g. modeling of latency or modeling of cost
- Papers that are not focused on smart contracts
- Modeling solutions that only allow modeling a very specific set of use cases (e.g. only financial contracts)
- Modeling solutions that are not intuitive or not understandable by people who are not blockchain experts e.g. text-based methods

2.4 Papers Selection

The first step of the selection was to look in digital libraries using the previously mentioned search terms. A lot of results were generated from these searches and not all of them were actually relevant. The search results were then downloaded as BibTeX files and a python script was written to filter the results. The python script simply went through each result and verified that the search terms were either in the title or in the abstract. After running the script on the BibTeX files of the four main sources, the following number of articles remaining is shown for each source in the first column of Table 1. The abstract of each of these articles was read to carry out an additional phase of filtering. Finally, the remaining articles were downloaded, read, and filtered based on the including and excluding criteria. The final number of articles for each source is

shown in the second column of Table 1. The last column of Table 1 shows the articles that were found through related work sections and citations of the first articles. Figure 1 shows the number of articles at each step of the identification of relevant articles for the literature review.

Table 1. Number of identified papers by source

	Initial search	**Inclusion criteria**	**Snowballing**
IEEE	20	2	0
Science Direct	45	0	0
ACM	31	4	0
Scopus	141	2	0
Springer	N/A	N/A	5
Wiley	N/A	N/A	1
Elsevier	N/A	N/A	1

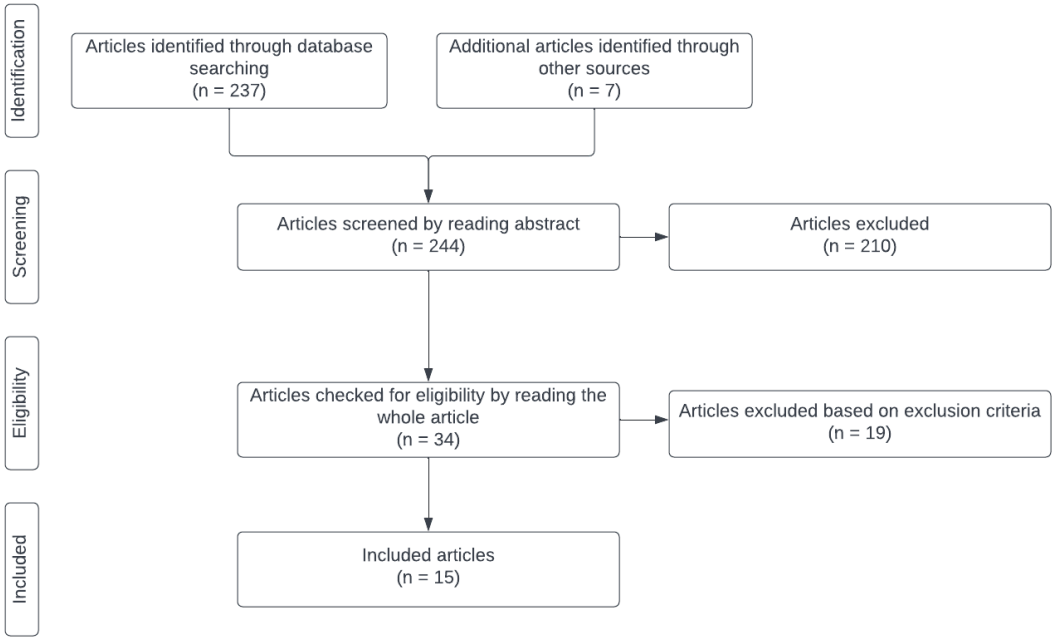

Figure 1. Steps for identification of literature review papers

2.5 Information Extraction

The blockchain application elements that are modeled through the solutions suggested in the identified papers are not known in advance. This means that the list of elements is built along the way as the articles are read. For each modeling solution, the objective is to determine which elements are modeled and how they are modeled. Some elements might be modeled by a solution even if it is not mentioned in the article. If that is the case, then the element will be marked as not modeled, because it is not realistic to go explore in detail every single modeling solution.

2.6 Summary of Selected Articles

In this section, each publication that was selected as a result of the papers' selection step of the literature review is summarized. Some approaches have similar contributions, so they have been grouped together. For example, [MBH18] is an extension of [LPGBD$^+$19], so both of these contributions are described in the same paragraph.

Rocha et al. "Preliminary steps towards modeling blockchain-oriented software" [RD18]
In this article, no novel model-driven engineering method is proposed. This article shows how existing model methods could be used to model blockchain applications. More specifically, UML, BPMN, and ER models are used independently to try to model the main blockchain elements. The objective is to show how these approaches are lacking certain concepts to properly cover the different blockchain elements.

Mavridou et al. "Designing secure Ethereum smart contracts: A Finite State Machine based approach" [ML18]
The authors model smart contracts as finite state machines through a graphical editor. The idea is to use rigorous semantics to lay the foundation for formal verification tools. Smart contract (Solidity) code can be generated automatically from the model. The model also allows developers to easily integrate design patterns like access control into the smart contract.

Lopez-Pintado et al. "Caterpillar: a business process execution engine on the Ethereum blockchain" [LPGBD$^+$19] and Mercenne et al. "Blockchain Studio: A Role-Based Business Workflows Management System" [MBH18]
These articles look at blockchain as a means to support business processes. Lopez-Pintado et al. suggest building a BPMN-compatible business process management system (BPMS) on the Ethereum blockchain. The BPMN model is translated to multiple smart contracts that allow the execution and management of the business process. The tool comes with a user interface that facilitates the lifecycle management of the processes. Mercenne et al. create a fork of Caterpillar to support the management of roles and

access control.

Silva et al. "Decentralized Enforcement of Business Process Control Using Blockchain" [SGS18] and *Hornkov et al. "Exploring a Role of Blockchain Smart Contracts in Enterprise Engineering" [HSP18]*
Both of these works use the Enterprise Ontology DEMO to model blockchain applications. Silva et al. propose a translation between DEMO and the blockchain platform Hyperledger. The objective is to improve the control and traceability of collaborative business processes. The method presented by Hornácková et al. suggests using an Enterprise information system (EIS) with a blockchain system. The EIS supports the transactions that do not need to be stored on the blockchain and communicates with the blockchain through an API for the other transactions. The presented method does not offer automatic code generation.

Ladleif et al. "Modeling and Enforcing Blockchain-Based Choreographies" [LWW19], Corradini et al. "Engineering trustable choreography-based systems using blockchain" [CMM+20] and Weber et al. "Untrusted Business Process Monitoring and Execution Using Blockchain" [WXR+16]
Ladleif et al. suggest extending BPMN choreographies to support business processes on the Ethereum blockchain platform. Choreography diagrams look at the system from a message exchange perspective. In this case, blockchain technology allows different participants to verify that a particular message has been sent at a given time and that the messages that are sent respect the predefined conditions of the model. As an example of an extension that Ladleif et al. make to choreographies, the concept of data objects and script tasks are added to choreography diagrams to better represent blockchain applications. Similarly, Corradini et al. suggest a method that uses BPMN choreography diagrams to model blockchain applications. This method also offers the possibility to search through a repository of choreographies which enhances reusability and a user interface to manage the lifecycle of the application. Weber et al. also use choreography diagrams to ensure that only conforming messages advance the state of the process. One of the main differences with the solution of Corradini et al. is that the model goes into more detail and that more components are deployed on the blockchain. For instance, in the last approach, individual processes that only involve one participant are modeled, whereas these processes are abstracted in the second approach.

Marchesi et al. "An Agile Software Engineering Method to Design Blockchain Applications" [MMT18]
The main contribution of this paper is to suggest an Agile method that separates the development of an application into two parts: the smart contracts and the more traditional software system. It suggests a series of steps to ensure a rigorous development process.

The idea is to consider at the design stage all of the components of the whole application, from the user interface to the external smart contracts that the blockchain part of the application may have to call, putting the emphasis on the security concerns that come with both traditional and blockchain applications. The method in this paper does not allow automatic code generation.

Hamdaqa et al. "iContractML" [HMQ20]
These authors first do a feature-oriented domain analysis of three common blockchain platforms: Hyperledger, Azure Blockchain Workbench, and Ethereum to come up with a reference model for smart contracts. These platforms have been chosen because they represent the different types of blockchain platforms. These types of platforms are permissioned, blockchain as a service, and public platforms. The reference model is then realized as a modeling framework, a platform-independent modeling language called iContractML, that allows automatic code generation. The validity of the model can be ensured through validation rules written in Acceleo Query Language.

Garamvolgyi et al. "Towards Model-Driven Engineering of Smart Contracts for Cyber-Physical Systems" [GKGK18]
This article explores the use of UML statecharts to model cyber-physical systems (CPS). More specifically, the logic implemented as a smart contract is used as a digital twin to the real-life CPS. While at the time of writing only Ethereum is supported, the approach is meant to be generic enough to eventually allow generating blockchain code from different platforms.

Lu et al. "Integrated model-driven engineering of blockchain applications for business processes and asset management" [LTW+20] In this article, the authors present a tool called Lorikeet that allows the modeling of business processes and digital assets. They allow the creation of both fungible and non-fungible asset registries, as well as escrow and asset swap. The tool comes with a user interface to facilitate the modeling and interaction with the blockchain application.

Babkin et al. "Model-Driven Liaison of Organization Modeling Approaches and Blockchain Platforms" [BK20]
In this paper, the authors suggest a mapping between concepts of ArchiMate enterprise architecture modeling language and the HyperLedger Composer modeling language. The authors explain the use of Archimate by arguing that some languages such as BPMN cannot be considered as a desired modeling approach because as opposed to ArchiMate, they are not complete organizational models. The implementation of the automatic code generation is done in Python and the code translation is done semi-automatically i.e. the business logic must be manually written.

Boubeta-Puig et al. "CEPChain: A graphical model-driven solution for integrating complex event processing and blockchain" [BPRBM21]
CEPchain connects a Complex Event Processing (CEP) system to the Ethereum blockchain platform. More specifically, it allows the modeling of the automatic trigger of smart contract transactions when event pattern conditions are met. It supports the deployment of smart contracts onto the blockchain platform and supports loading BPMN models of smart contracts generated by Caterpillar.

2.7 Presentation of Results

In this section, the results of the literature review are presented in the form of tables. Table 2 contains the elements aiming at answering research question **[RQ1.1] What are the main characteristics of existing model-driven engineering solutions?** The characteristics found in the table are the following:

Table 2. Characteristics of modeling solutions

Authors	Platform	Modeling solution	Formal modeling	Code generation	Data privacy	Lifecycle management	Open source
Corradini et al. [CMM+20]	Ethereum	BPMN choreography	No	Full	No	Yes	Yes
Marchesi et al. [MMT18]	Ethereum	UML (class, sequence) extended with stereotypes	No	No	No	No	No
Hamdaqa et al. [HMQ20]	Azure workbench, Ethereum, Hyperledger	DSL (eCore)	No, but has model validation rules	Partial	No	No	Yes
Garamvölgyi et al. [GKGK18]	Ethereum	UML statecharts (Yakindu)	No	Partial	No	No	No
Babkin et al. [BK20]	Hyperledger	Archimate	No	Partial	Yes	No	No
Lu et al. [LTW+20]	Ethereum	BPMN	No	Full	No	Yes	No
Rocha et al. [RD18]	Ethereum	BPMN, ER and UML	No	No	No	No	No
Weber et al. [WXR+16]	Ethereum	BPMN choreography	No	Full	Yes	No	No
Ladleif et al. [LWW19]	Ethereum	BPMN choreography	No	Full	No	No	No
Hornáčková et al. [HSP18]	Ethereum	DEMO	No	No	No	No	Yes
Silva et al. [SGS18]	Hyperledger	DEMO	No	No	Yes	No	No
López-Pintado et al. [LPGBD+19]	Ethereum	BPMN	No	Full	No	Yes	Yes
Mercenne et al. [MBH18]	Ethereum	BPMN	No	Partial	No	Yes	No
Mavridou et al. [ML18]	Ethereum	FSolidM	Not yet	Full	No	No	Yes
Boubeta-Puig et al. [BPRBM21]	Ethereum	BPMN and EMF	No	Full	No	Yes	No

Platform The blockchain platform whose elements are modeled by the solution.

Modeling solution The model type or the model language that is used in the solution.

Format modeling The possibility to create models with formal semantics or transform models into models with formal semantics. Such models can be unambiguously verified, meaning that the model verification provides guarantees that certain characteristics of the model will remain true in all situations. Examples of formal verification methods are theorem proving and model checking [MA19].

Code generation The degree to which the model can be automatically transformed into source code, and whether the generated code can be deployed as-is or need manual additions/modifications. This characteristic does not include code generation for oracles, because this is evaluated separately in another research question.

Data privacy The possibility to restrict access to the information stored on the blockchain.

Lifecycle management The support of the lifecycle of the blockchain application, such as a method to facilitate the deployment of the application on the blockchain or the interaction with the smart contracts.

Open source The possibility to have access to the source code of the solution. This is false by default if a modeling solution has never been implemented and is simply suggested. This is false if a new modeling solution is suggested and implemented, but it is not made available. This is true if using an existing modeling solution and the models presented are made available.

Tables 3 and 4 contain the elements necessary to answer research question **[RQ1.2] What are the main elements of blockchain applications that are found in existing modeling solutions?** The main blockchain elements that were identified are the following:

Smart contract logic Set of elements that can be used to generate the function body of smart contracts and determine the behavior of the application.

Smart contract data The type and the value of the information that will be stored in the smart contract (on the blockchain).

Third party smart contract Smart contracts that are not directly part of the modeled blockchain application. It could be any smart contract that existed before the application was modeled.

Table 3. Blockchain elements of modeling solutions part 1

Authors	Smart contract logic / business rules	Smart contract data	Third party smart contract	Event
Corradini et al. [CMM+20]	Gateways and guards, sequence flows	Partly represented by message annotations	***	Start event, end event
Marchesi et al. [MMT18]	UML classes and stereotypes	UML class attributes and stereotypes	UML classes and stereotypes	***
Hamdaqa et al.	UML classes	UML classes	***	UML class
Garamvölgyi et al. [GKGK18]	Transition, guards	State, composite state, history	***	***
Babkin et al. [BK20]	***	Artifact, BusinessObject, DataObject	***	ApplicationEvent, BusinessEvent, ImplementationEvent, TechnologyEvent
Lu et al. [LTW+20]	BPMN scripts	Script tasks, asset template forms	SmartContract Interface icon	BPMN events, UI event monitor
Rocha et al. [LTW+20]	ER: ***, UML: ***, BPMN: derived from model	ER logical design: database columns, UML: class attributes, BPMN: ***	***	***
Weber et al. [WXR+16]	BPMN choreography	***	***	BPMN events
Ladleif et al. [LWW19]	BPMN script tasks	BPMN data objects	Script tasks	***
Hornáčková et al. [HSP18]	DEMO actions	Transaction object facts	***	***
Silva et al. [SGS18]	***	DEMO Fact	***	***
López-Pintado et al. [LPGBD+19]	BPMN scripts and deduced from control flow of BPMN diagram	***	***	BPMN events
Mercenne et al. [MBH18]	BPMN scripts and deduced from control flow of BPMN diagram	***	***	BPMN events
Mavridou et al. [ML18]	***	Variables in code editor	***	***
Boubeta-Puig et al. [BPRBM21]	BPMN scripts and deduced from control flow of BPMN diagram	***	***	BPMN events

Table 4. Blockchain elements of modeling solutions part 2

Authors	Transaction	Participant	Roles and permissions	Asset
Corradini et al. [CMM+20]	Tasks (messages)	Initiator and recipient of messages	Checkboxes when creating instance of choreography	Message annotations
Marchesi et al. [MMT18]	UML sequence	UML stereotype	UML stereotype	New message in UML sequence
Hamdaqa et al. [HMQ20]	UML classes and relationships	UML class	UML classes and relationships	UML class
Garamvölgyi et al. [GKGK18]	Action, Transition	***	Transaction guard expressions	***
Babkin et al. [BK20]	Application Interaction, BusinessInteraction, TechnologyInteraction	BusinessActor, Stakeholder, ApplicationComponent	Access Relationship	Same as smart contract data
Lu et al. [LTW+20]	BPMN task	User tasks	***	Asset template forms
Rocha et al. [RD18]	ER logical design: ***, UML: class function, association, BPMN: ***	ER: *** , UML: smart contract class, BPMN: lane	***	ER logical design: database column, UML: class attribute, BPMN: ***
Weber et al. [WXR+16]	Choreography messages	Lanes	Lanes and message sender and receiver information	***
Ladleif et al. [LWW19]	Sequence flow, message flow	Initiator and recipient of messages	inferred from sequence flow	***
Hornáčková et al. [HSP18]	DEMO transaction	DEMO actor	DEMO Actor Function Matrix	***
Silva et al.	Business transaction step	DEMO actor	DEMO actor role	Busines transaction instance
López-Pintado et al. [LPGBD+19]	BPMN tasks	BPMN lanes	***	***
Mercenne et al. [MBH18]	BPMN tasks	BPMN lanes	BPMN task role attribute	***
Mavridou et al. [ML18]	State transition	***	Guards	Variables in code editor
Boubeta-Puig et al. [BPRBM21]	BPMN tasks	BPMN lanes	***	***

Event An event emitted on the blockchain that other actors can subscribe to.

Asset Something of value that can be exchanged. For example, in Ethereum, a fungible or non-fungible token.

Transaction An action that modifies the state of the blockchain and/or the state of the application.

Participant An actor that interacts with the application.

Roles and permissions Roles assign responsibilities to actors and permissions allow certain roles to execute certain actions. For instance, the owner of a smart contract could be the only actor allowed to empty the balance of that smart contract.

Finally, Table 5 answers research question **[RQ1.3] What types of oracles are modeled and how are they modeled?** Four types of oracles have been chosen to form the basis for answering [RQ1.3]. Note that to our knowledge, these four types of oracles were described for the first time in [MBF+20]. Below is a description of the four types of oracles. Also note that in Table 5, not all solutions are represented. If a solution is missing, it means that it neither models nor automatically generates code for blockchain oracles.

Rocha BPMN: Modeled using a combination of sequence flows

Table 5. Oracle coverage of the MDE solutions

Authors	Evaluation criteria	Pull-based inbound	Push-based inbound	Pull-based outbound	Push-based outbound
Corradini et al. [CMM+20]	Modeled	No	No	No	No
	Automatic generation	No	Partially	Yes	No
Babkin et al. [BK20]	Modeled	No	No	No	No
	Automatic generation	No	Partially	Yes	No
Lu et al. [LTW+20]	Modeled	Partially modeled through events	No	No	Partially modeled through events
	Automatic generation	No	Partially	Yes	No
Rocha et al. (BPMN) [RD18]	Modeled	Yes	Yes	Yes	Yes
	Automatic generation	No	No	No	No
Rocha et al. (UML) [RD18]	Modeled	No	Yes	Yes	No
	Automatic generation	No	No	No	No
Weber et al. [WXR+16]	Modeled	Partially modeled through events	No	No	Partially modeled through events
	Automatic generation	No	No	No	No
Ladleif et al. [LWW19]	Modeled	No	No	No	No
	Automatic generation	No	Partially	Yes	No
Lopez-Pintado et al. [LWW19]	Modeled	Partially modeled through events	No	No	Partially modeled through events
	Automatic generation	No	Partially	Yes	No
Boubeta-Puig et al. [BPRBM21]	Modeled	No	Yes	Partially	No
	Automatic generation	No	Yes	Partially	No

Pull-based inbound: In this type of oracle, a smart contract initiates the interaction and the data flows from an off-chain application to the blockchain. An off-chain component subscribes to events on the blockchain. Events are used by on-chain requesters to fetch

information from outside the blockchain. Once the oracle has detected an event from a smart contract, it will fetch the data and send a signed transaction containing the data to the requesting smart contract. These oracles are useful when a blockchain component needs to access data once in a while at a specific moment. For instance, this could be used to verify a buyer's creditworthiness once an order is made by the buyer.

It is worth noting that the implementation of this type of oracle can vary. In one case, the requested data is stored in an oracle smart contract that is regularly updated. This architecture can be called a storage oracle [LWW20]. Another way to implement this is to return the data directly to the requesting smart contract, meaning that a callback is specified when requesting the data. The callback represents a function of a smart contract that is called by the oracle with the requested data. This architecture can be called a request-response oracle [LWW20].

Push-based inbound: In this type of oracle, the initiator is an off-chain component and the data flow is from an off-chain application to the blockchain. An off-chain component is configured to query the status of an off-chain component and send data under certain conditions. The data is then translated into a blockchain transaction and sent on the blockchain. For example, a push-based inbound oracle could be used to send the notification that a package has been delivered along with the parcel's information.

Pull-based outbound: In this type of oracle, the initiator is an off-chain component and the data flow is from the blockchain to an off-chain application. For example, a web application could query the status of the production of a specific good.

Push-based outbound: In this type of oracle, the initiator is an on-chain component and the data flow is from the blockchain to an off-chain application. For example, a notification could be sent from a smart contract to a web application once an event listener has detected that a package has been delivered.

Table 5 indicates for each solution if the automatic generation of code is fully supported, partially supported, or not supported. However, for some solutions, this table would benefit from additional details. For this reason, we use the end of this section to provide those additional details. The partial automatic generation of push-based inbound of [CMM+20] means that since a UI is generated by the solution, it is possible to manually send information to the blockchain through that interface. However, it does not allow querying off-chain applications so it cannot be considered as generating the complete push-based inbound oracle. The partial automatic generation of push-based inbound oracle for [BK20] means that the REST API of Hyperledger can be used to send data to the blockchain. However, there is no support for querying off-chain applications so it does not generate the complete push-based inbound oracle. The partial

automatic generation of push-based inbound oracle for [LTW+20] is similar to the two previous solutions. An automatically generated API provides the possibility to update the blockchain, but cannot monitor off-chain applications. Additionally, this solution supports pull-based outbound oracles through its API. There is nothing to add about the code generation of [RD18] (UML), since it does not support any level of code generation. The same goes for [WXR+16]. The automatic generation of the push-based inbound oracle in [LWW19] is partially done through the use of a custom software interface that can be generated from the choreography model. Similarly, a REST API is automatically generated in [LPGBD+19] which constitutes part of the push-based inbound oracle. Additionally, this last solution fully supports the generation of pull-based outbound oracles through their REST API. Finally, [BPRBM21] suggests an interesting approach for oracles. This solution can generate a complex event processing system that listens to off-chain events that trigger smart contract transactions on certain conditions. This is how the push-based inbound oracle is fully automatically generated. This solution also potentially supports the pull-based outbound oracle. It can be used to make smart contract transactions, which is the first step of the pull-based outbound oracle. However, it is not mentioned whether the data returned by the function can be forwarded to external applications.

2.8 Answers to Research Questions

The results of the literature review have been presented in Section 2.7. We can use and interpret these results to find the answer to research question 1: **[RQ1] What are the strengths and weaknesses of the model-driven engineering solutions for blockchain-based applications?** To do that, for the identified model-driven solutions, the strengths and weaknesses of the characteristics (RQ1), the blockchain elements (RQ2) and the oracle coverage (RQ3) are described.

[RQ1.1] What are the main characteristics of existing model-driven engineering solutions?

Only 3 out of 15 of the examined modeling methods allow data privacy. Generally speaking, solutions targeting public blockchains like Ethereum do not support data privacy since this is an inherent limitation of the platform. None of the methods allows formal modeling, even though [ML18] is designed with that purpose. Formal modeling can be useful to unambiguously verify a model against a set of validation rules. There do exist blockchain modeling solutions that allow formal modeling, however, they are mostly text-based and the choice was made in this research to focus on visual or intuitive modeling solutions. 5 out of 15 solutions are open source, but the source code and/or artifacts of the solutions could possibly be made available by the authors upon request. Finally, only 2 out of 15 solutions model blockchain applications that are not hosted on the Ethereum platform, which means that some platforms like Hyperledger or Corda are

not very well covered or not covered at all by the listed solutions.

From Table 2, we can see that 11 out of 15 solutions support partial or full code generation. 5 out of 15 solutions support lifecycle management of the application, meaning that they help the developer with the deployment of the smart contracts, as well as with the interaction with the smart contracts through user interfaces. Finally, we can see that popular modeling solutions such as BPMN and UML have been reused in the context of blockchain applications which contributes to their adoption and usability.

[RQ1.2] What are the main elements of blockchain applications that are found in existing modeling solutions?

The interaction with third-party smart contracts is modeled by only 3 out of 15 solutions. This is a big limitation since smart contracts will often need to access data from other contracts on the blockchain. Additionally, as shown in Table 4, the assets that can be managed on the blockchain are represented in 8 out of 15 existing solutions. It turns out that the methods which provide code generation functionality usually do not model assets. This is a limitation for anyone interested in creating tokens or interacting with different tokens. Roles and permissions are also covered by 8 out of 15 solutions. Finally, blockchain events are covered by 8 out of 15 solutions.

As we can see in Table 4, blockchain transactions are covered by 15 out of 15 solutions. They are the elements that have been the most examined and they are well modeled by the different solutions. The participants, the smart contract logic and business rules, and the data stored in the smart contract also seem to be relatively well modeled. They are covered by 13 out of 15, 12 out of 15, and 11 out of 15 respectively.

[RQ1.3] What types of oracles are modeled and how are they modeled?

According to the literature review, even though there have been a few attempts at modeling pull-based inbound and push-based outbound oracles, there is no modeling solution that can automatically generate them. There is also no solution among the ones identified in the literature review that can model the four types of oracles described in this thesis.

In a few solutions, push-based inbound and pull-based outbound oracles are automatically generated through a user interface. This means that the user can conveniently get data from the blockchain application and make transactions to smart contracts.

3 Requirements and Design

Based on the answers to the research questions of the literature review found in Section 2.8, we can see that one of the limitations of the existing literature is the lack of proper modeling solutions for blockchain oracles. We believe that oracles are an important element of blockchain applications since they allow the communication between blockchain applications and traditional off-chain applications. For this reason, we decided to base the contribution of this thesis on that topic and suggest a model-driven engineering solution for blockchain oracles. To achieve this, we first identify the goals of the systems and the users who will likely find this contribution useful. Then we find some use cases for the contribution, from which we derive some functional requirements, security requirements, and their associated acceptance criteria. In this section, we answer the research question **[RQ2] What are the requirements for a model-driven engineering solution for blockchain oracles?** To better answer this question, we divide it into three sub-research questions. The first sub-research question is **[RQ2.1] What are the goals of the model-driven engineering solution?** The second sub-research question is **[RQ2.2] What are the use cases of the model-driven engineering solution?** The third sub-research questions is **[RQ2.3] What are the functional and security requirements of the model-driven engineering solution?**

3.1 System Goals

The literature review shows that there currently is no model-driven engineering solution that supports the four types of oracles described previously. The objective of this thesis is to address this limitation and develop a plugin that can model four types of oracles and that can automatically generate the source code from the model. The generated source code should be a software application that runs independently of any other application and that can be used by another application to query and modify the state of the blockchain. This way, regular off-chain applications do not need to know how to interact with the blockchain directly, and they delegate that work to the oracle application through HTTP requests. The Ethereum blockchain has been chosen as the blockchain to be used by the application generated from the model because it is a mature blockchain platform that is heavily used to create decentralized applications and which does not come with default functionality for querying data that is stored off-chain [XPZ+18]. The goals of the contribution of this thesis are listed and described in Table 6.

3.2 Targeted Users

In this section, the main characteristics of the typical users of the plugin and oracle application developed in this thesis are described.

Table 6. System goals

Goal ID	Goal description
G1	Provide an intuitive model for oracles with domain-specific constructs (UML profile)
G2	Provide a model verification feature to check that the model is valid before the source code can be generated.
G3	Define transformation rules to map elements of the model to source code elements.
G4	Design a source code generation feature to take the model and generate the source code.
G5	Generate a standalone application from the model that can be used for listening to events on the blockchain, querying the blockchain, and changing the state of the blockchain through HTTPS requests.

Needs access to blockchain The main goal of the user of the oracle is to establish communication between a blockchain application and an off-chain application. It can be that the off-chain application needs data from the blockchain and/or the blockchain application needs access to external data.

Not a blockchain expert Blockchain applications can be very complex and it is not necessarily intuitive to interact with them. The user wants to be able to communicate with the blockchain without having to write too much source code and without having to know which libraries to use. The user wants to have an intuitive graphical model to help describe and document that oracle. The graphical model should be for the most part understandable by people who are not blockchain experts.

IT aware Since the user needs to establish communication between a blockchain application and an off-chain application, the user has a basic understanding of information technologies, has access to an existing web application, and is able to make basic changes to it to send and receive HTTPS requests.

3.3 Use Cases

In this section, the different use cases describe how the plugin and generated application are meant to be used by users and which goals enable these scenarios. The use cases are separated into two diagrams. The first diagram relates to the modeling and code generation part that precedes the execution of the oracle application. These are the use cases that belong in the system boundary of the MagicDraw plugin. The second diagram shows how the oracle application will be used by off-chain applications. These are the

use cases that belong to the system boundary of the oracle application and the smart contracts on the blockchain.

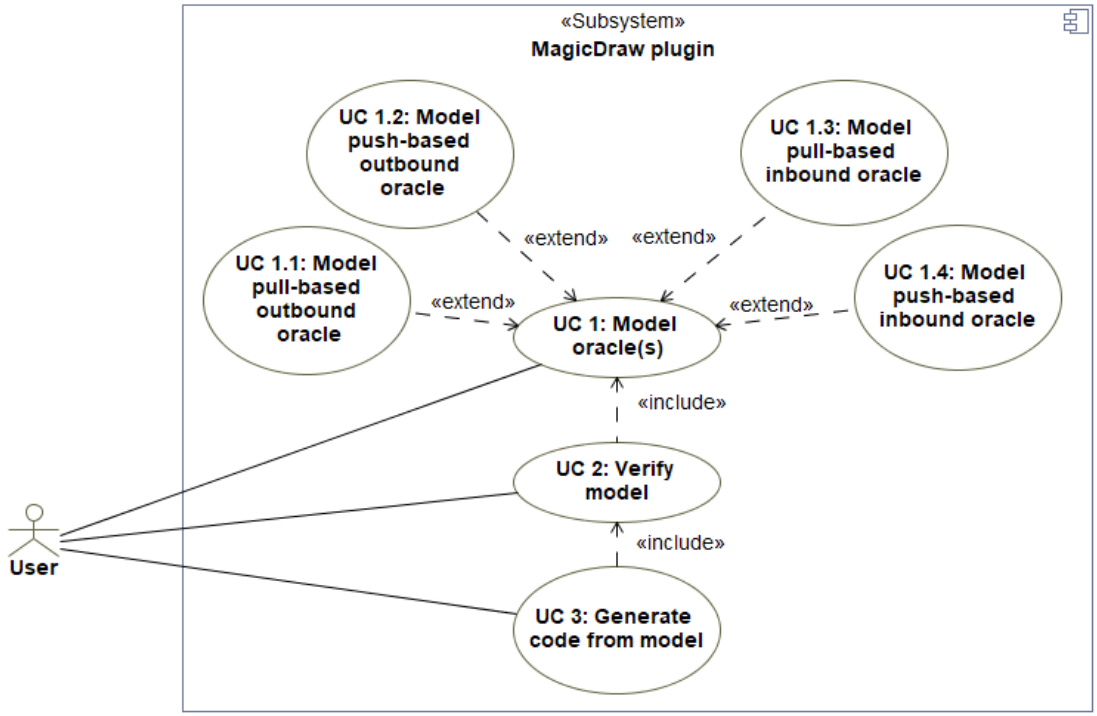

Figure 2. Use case diagram for MagicDraw plugin

3.3.1 Oracle Modeling

The first use case is represented in Table 7. In this use case, a user creates a model of one or more oracles. The whole model can be composed of one or more models of each type of oracle. In order to be able to use the tool, the user must have installed the MagicDraw plugin and must have access to the Oracle UML profile in order to apply the right stereotypes and tagged values to the elements of the sequence diagram. Since the oracles are modeled using sequence diagrams, the user must first create all required lifelines and assign their stereotypes attributes. The same goes for the messages and the user must make sure that they are added in the correct order.

3.3.2 Verify Model

The second use case is represented in Table 8. In this use case, the user triggers a model verification step before generating the source code. The objective is to make sure that no

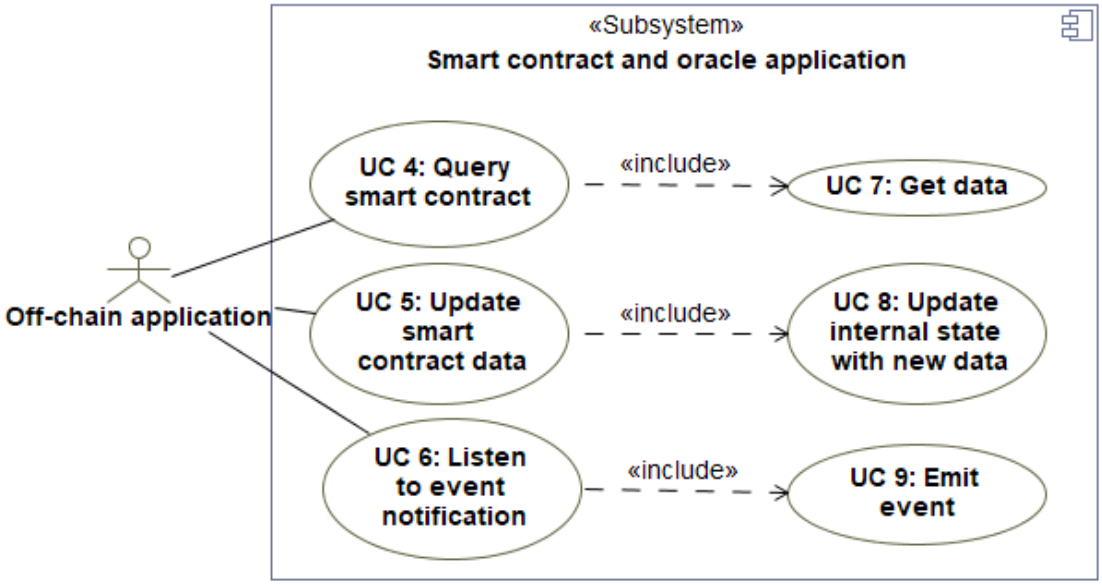

Figure 3. Use case diagram for smart contract and oracle application

Table 7. Use case 1: Oracle Modeling

Use case #	UC1
System goal #	G1
Use Case Name	Oracle Modeling
Actors	User
Trigger	User needs to model one or more oracles.
Preconditions	1. User has access to Oracle model elements. 2. User has installed the Oracle plugin in MagicDraw.
Postconditions	1. Oracle(s) are modeled.
Normal flow	1. User creates diagram. 2. MagicDraw shows a new blank diagram. 3. User opens Oracle UML profile. 4. MagicDraw shows Oracle elements. 5. User drags elements to diagram and applies stereotypes. 6. Diagram shows elements with their respective stereotypes. 7. User adds tagged values to stereotypes. 8. MagicDraw shows the tagged values of the elements in the diagram.

invalid oracle code is generated. The precondition is that there needs to be a model to be verified. If the model is valid, then the model verification passes, and the plugin will proceed to the source code generation, which is the next use case. Otherwise, the plugin will show an error message indicating the nature of the problem with the model and the code generation will not start until the problem(s) are fixed in the model.

Table 8. Use case 2: Verify Model

Use case #	UC2
System goal #	G2
Use Case Name	Verify Model
Actors	User
Trigger	User wants to verify that the model does not contain errors.
Preconditions	User has modeled one or more oracles.
Postconditions	The modeled oracles have been verified and the plugin is ready for next use case.
Normal flow	1. User clicks on code generation button. 2. Plugin verifies model successfully. 3. Plugin shows no error message.
Alternative flow	1. User clicks on code generation button. 2. Plugin verifies the model and detects problems in the model. 3. Plugin shows an error message indicating which element in the model is incorrect.

3.3.3 Generate Code from Model

The third use case is represented in Table 9. In this use case, the user generates the code from the model. The precondition for this action to be taken is that the model verification succeeds. To start the code generation, the user simply has to click on the right button in the user interface of MagicDraw. Afterward, the source code is generated at the path specified in the model. This is the last use case that belongs to the MagicDraw plugin subsystem. The following use cases belong to the oracle and smart contract subsystem.

3.3.4 Query Smart Contract

The fourth use case is represented in Table 10. This is the first use case that belongs to the oracle and smart contract subsystem. This use case and the following ones depend on the successful source code generation from the use cases of the MagicDraw plugin subsystem. What is referred to as the oracle in the following use cases is the source code

resulting from the code generation. In the fourth use case, an off-chain application makes an HTTPS request to the oracle to query the state of the blockchain.

Table 9. Use case 3: Generate Code from Model

Use case #	UC3
System goal #	G3 and G4
Use Case Name	Generate Code from Model
Actors	User
Trigger	User wants to generate oracle source code from the model.
Preconditions	Model has been verified.
Postconditions	Source code is generated.
Normal flow	1. User clicks on code generation button. 2. MagicDraw shows a dialog box with a message indicating that the code generation was successful.

Table 10. Use case 4: Query Smart Contract

Use case #	UC 4
System goal #	G5
Use Case Name	Query Smart Contract
Actors	Off-chain application
Trigger	Oracle receives a request for querying a smart contract's data from an off-chain application.
Preconditions	1. The oracle application is running. 2. The off-chain application making the request owns a valid certificate trusted by the oracle application.
Postconditions	1. A reply containing the requested data has been sent to the off-chain application by the oracle.
Normal flow	1. Oracle authenticates off-chain application. 2. Oracle verifies parameters from HTTPS request. 3. Oracle gets requested data from the smart contract by calling the view function specified in the request. 4. Oracle returns data to the off-chain application.
Alternative flow	1. Oracle fails to authenticate the off-chain application or oracle detects invalid parameters in HTTPS request. 2. Oracle returns an error message to the off-chain application.

3.3.5 Update Smart Contract Data

The fifth use case is represented in Table 11. This use case is similar to the fourth one. The key difference here is that the off-chain application sends data to be stored on the blockchain instead of simply requesting information from the blockchain. Note that in this use case, there are two alternative flows to the first step of the normal flow. In the first alternative flow, the sequence is not initiated by the off-chain application, but by the oracle. The oracle queries the off-chain application for data to be sent to the blockchain. When the off-chain application replies with data to be sent, the rest of the flow continues as usual. This alternative flow is an alternative way of implementing the push-based inbound oracle. In the second alternative flow, the request to update the smart contract data is made as a response to an event notification sent by the oracle. This alternative flow would happen in the context of a pull-based inbound oracle, where the smart contract emits an event in order to get some data sent to it.

3.3.6 Listen to Event Notifications

The sixth use case is represented in Table 12. In this use case, the application listens to events emitted on the blockchain. Because the off-chain application does not know how to interact with the blockchain directly, the oracle must be configured to do that job for the off-chain application and subscribe to the events that the off-chain application is interested in.

3.3.7 Get Data

The seventh use case is represented in Table 13. This use case represents the actions of the smart contract when one of its call functions has been called by the oracle. As shown in Figure 3, this use case follows use case 4. Note that in the functional requirements section, this use case has no associated requirements. This use case is simply there to show how the oracle will interact with the blockchain, but since the smart contract code is out of the scope of the contribution of this thesis and cannot be generated from the model, there are no associated requirements.

3.3.8 Update Internal State with New Data

The eighth use case is represented in Table 14. This use case represents the actions of the smart contract when one of its functions has been called by the oracle to update the data of the smart contract. As shown in Figure 3, this use case follows use case 5. Note that in the functional requirements section, this use case has no associated requirements. This use case is simply there to show how the oracle will send data to the blockchain, but since the smart contract code is out of the scope of the contribution of this thesis and cannot be generated from the model, there are no associated requirements.

Table 11. Use case 5: Update Smart Contract Data

Use case #	UC 5
System goal #	G5
Use Case Name	Update Smart Contract Data
Actors	Off-chain application
Trigger	Oracle receives a request to update a smart contract's data from an off-chain application
Preconditions	1. The oracle application is running. 2. The off-chain application owns a valid certificate trusted by the oracle application.
Postconditions	1. The state of the smart contract has been updated.
Normal flow	1. Oracle authenticates off-chain application. 2. Oracle verifies parameters from HTTPS request. 3. Oracle updates smart contract with the data provided by the off-chain application by calling the function specified in the request. 4. Oracle returns transaction confirmation to off-chain application.
Alternative flow	1 (b) 1. Oracle queries off-chain application for data to send to the smart contract. 2. Off-chain application returns data to be sent to the smart contract. 1 (c) 1. Oracle sends an event notification to an off-chain application. 2. Off-chain application replies with the transaction request.

3.3.9 Emit Event

The ninth use case is represented in Table 15. This use case represents the action of emitting an event by the smart contract. As shown in Figure 3, this use case follows use case 6. Note that in the functional requirements section, this use case has no associated requirements. This use case is simply there to show that the events that the off-chain application listens to are initiated by the smart contracts on the blockchain, but since the smart contract code is out of the scope of the contribution of this thesis and cannot be generated from the model, there are no associated requirements.

Table 12. Use case 6: Listen to Event Notifications

Use case #	UC 6
System goal #	G5
Use Case Name	Listen to Event Notifications
Actors	Off-chain application
Trigger	Smart contract emits an event.
Preconditions	1. Smart contract exists on the blockchain. 2. Oracle has subscribed to the smart contract event.
Postconditions	1. The off-chain application has been notified of the event.
Normal flow	1. Oracle detects the event. 2. Oracle transforms the event into an HTTPS request containing the data of the event and sends it to the off-chain application.

Table 13. Use case 7: Get Data

Use case #	UC 7
System goal #	G5
Use Case Name	Get Data
Actors	Smart contract
Trigger	Smart contract receives a view function call.
Preconditions	1. Smart contract exists on the blockchain. 2. Smart contract has at least one view function.
Postconditions	1. The smart contract function has finished executing successfully.
Normal flow	1. Smart contract receives function call. 2. SC function executes its instructions. 3. SC function returns requested data.

3.4 Functional Requirements

In Table 16, the functional requirements are associated with one of the use cases listed in the previous section and describe what features must the system provide so that the users can achieve their goals through the different use cases. For example, use cases *1.x* are associated with use case 1, use cases *2.x* are associated with use case 2, and so on. The functional requirements aim at describing the features that the proposed solution should

Table 14. Use case 8: Update Internal State with New Data

Use case #	UC 8
System goal #	G5
Use Case Name	Update Internal State with New Data
Actors	Smart contract
Trigger	Smart contract function is called with parameters containing new data.
Preconditions	1. Smart contract exists on the blockchain. 2. Smart contract has at least one function that modifies its internal state.
Postconditions	1. The SC function has finished executing successfully and the data of the smart contract has been updated.
Normal flow	1. Smart contract receives function call. 2. SC function executes its instructions.

Table 15. Use case 9: Emit Event

Use case #	UC 9
System goal #	G5
Use Case Name	Emit Event
Actors	Smart contract
Trigger	Smart contract executes function which emits an event.
Preconditions	1. Smart contract exists on the blockchain. 2. Smart contract has at least one function that emits an event.
Postconditions	1. An event is emitted on the blockchain that can be seen by anyone.
Normal flow	1. Smart contract executes function call. 2. Smart contract populates event data and emits the event.

have. Functional requirements 1.1 to 1.5 describe the features of the solution's UML profile (i.e. the model) and the elements that it should contain. Functional requirements 2.1 to 2.4 describe what is expected from the MagicDraw plugin to read the model into memory. Functional requirements 3.1 to 3.3 describe what is expected from the MagicDraw plugin to transform the model into source code. Functional requirements 4.1 to 4.4 describe the services and behavior of the oracle application generated from

the model. More specifically, they describe what is needed for the oracle application to answer data queries. Functional requirements 5.1 to 5.5 outline what is needed for the oracle application to be able to fulfill requests to update the blockchain. Finally, functional requirements 6.1 and 6.2 detail what is needed for the oracle application to successfully subscribe to blockchain events and notify the interested parties. Note that there are no functional requirements for the use cases described in Table 13, Table 14, and Table 15. The reason is that these use cases are there for clarity, but they are out of the scope of the contribution of this thesis.

Each functional requirement is associated with an acceptance criterion, which will be used in the evaluation section to make sure that the contribution of this thesis fulfills the functional requirements. The acceptance criteria are steps or actions that can be reproduced by anyone and that should be sufficient to confirm that the functional requirements have been successfully fulfilled. In this case, the acceptance criteria are elements that can be visually confirmed after performing a certain step. For example, acceptance criteria 1.1 to 1.5 target elements that can be seen in the MagicDraw application, like a UML profile or a UML sequence lifeline stereotype. Acceptance criteria 2.1 to 3.3 target elements that can be seen in the MagicDraw plugin or behaviors of the plugin that can be experienced while using the plugin in the MagicDraw application. Finally, acceptance criteria 4.1 to 6.2 target the source code and some aspects of the behavior of the generated oracle application, like HTTP requests responses.

3.5 Security Requirements

Table 17 describes the security requirements that must be satisfied by the model-driven engineering plugin so that the resulting generated code can achieve a certain level of security. These security requirements target the communication between the oracle application and off-chain applications. The reason why these security requirements have been chosen is that they target the most vulnerable part of the oracle application. One risk of blockchain oracles is that the data that they are supposed to bring to the blockchain can be compromised. By forcing the use of HTTPS between the oracle application and the off-chain applications, it ensures that the data sent between these two actors is encrypted and signed. The consequence of this is that the data cannot be modified and that the oracle application knows who it is communicating with. This reduces the probability that compromised data will be sent to the blockchain. For this purpose, the security requirements verify that valid X509 certificates have been configured in the model, verify their key size depending on the algorithm, and verify that the oracle application actually reads and checks those certificates. Each security requirement is associated with an acceptance criterion. The acceptance criteria verify the attributes of the X509 certificates using the open-source OpenSSL library. For verifying the behavior of the oracle application, the evaluation criteria describe what should happen in certain specific scenarios.

Table 16. Functional requirements and acceptance criteria

ID	Functional requirement	Acceptance criteria
1.1	The MagicDraw plugin must come with UML profile elements to model. oracles	The UML Profile can be opened using MagicDraw application.
1.2	The UML profile must have Lifeline stereotypes for actors involved in an oracle communication. oracles	The lifeline stereotypes for the actors are shown by expanding the UML Profile in MagicDraw.
1.3	The UML profile must have tag definitions for Lifeline stereotype attributes.	The tag definitions for the actors are shown by double-clicking on an actor lifeline stereotype in MagicDraw.
1.4	The UML profile must have Message stereotypes for messages involved in the oracle communication.	The lifeline stereotypes for the messages are shown by expanding the UML Profile in MagicDraw.
1.5	The UML profile must have tag definitions for Message stereotypes attributes.	The tag definitions for the messages are shown by double-clicking on a message lifeline stereotype in MagicDraw.
2.1	The MagicDraw plugin must read the modeled oracles from a MagicDraw model using the MagicDraw API.	MagicDraw API function calls can be seen by looking at the plugin's source code.
2.2	The MagicDraw plugin must verify that each oracle has a valid oracle type. This is based on the oracleType tagged value of the Bridge.	The MagicDraw plugin shows an error message when the user inputs an invalid oracle type or when the oracle type is missing.
2.3	The MagicDraw plugin must verify that an oracle contains the required stereotypes. This means that the required Messages and Lifelines should be in the diagram in the correct order based on the oracle type.	The MagicDraw plugin shows an error message if a stereotype of the model is missing, invalid, or in the wrong order. The error message indicates which element is missing or invalid and the reason why it is invalid.
2.4	The MagicDraw plugin must verify that the stereotypes' required tagged values are present.	The MagicDraw plugin shows an error message if a required tagged value is not present. The error message contains the name of the invalid tagged value and the reason why it is invalid.
3.1	The MagicDraw plugin must map elements of the model to elements in the source code. This mapping is based on the transformation rules.	The mapping between elements of the model and source code elements is visible in the plugin's source code.
3.2	The MagicDraw plugin must generate the source at the path specified in the model.	A single source code file named *oracle.js* is found at the path specified in the model after the code generation.
3.3	The MagicDraw plugin must not generate the source code if the model verification has failed.	There is no source code file generated at the specified directory after an error message is shown by the MagicDraw plugin. If there already was a source code file, then it remains unmodified.
4.1	The oracle application (the generated source code) must provide an API that can be used to call a view function of a smart contract.	A response is received by an application making a request to the oracle's API endpoint for blockchain queries.
4.2	The oracle application must match the parameters of the HTTPS request to the parameters of the smart contract function.	The mapping between an API request's parameters and the smart contract function call created by the oracle can be seen in the source code of the oracle.
4.3	The oracle must return a response with an error message when the request parameters are invalid or are missing.	A response with HTTP status 400 is returned to the client when the client makes a request with invalid parameters. The response also contains a *message* field with a description of the error.
4.4	The oracle must return the data that was requested.	The response from the oracle should contain the requested data.
5.1	The oracle application must provide an API that can be used to make a transaction and modify the state of the blockchain.	The */transaction* endpoint can be seen in the oracle source code and the mapping between an API request's parameters and the transaction created by the oracle can be seen in the source code of the oracle.
5.2	The oracle must accept transaction requests from replies to event notifications. These transactions are modeled through pull-based inbound oracles.	The response to an event notification is used to create the transaction and the mapping between the response content and the transaction created by the oracle can be seen in the source code of the oracle.
5.3	The oracle must accept transaction requests from replies to off-chain queries. These transactions are modeled in push-based inbound oracles.	The reply to the off-chain query is used to create the transaction and it can be seen in the source code of the oracle.
5.4	The oracle application must return a response with an error message when the request parameters are invalid are missing.	A response with HTTP status 400 is returned to the client when the client makes a request with invalid parameters. The response also contains a *message* field with a description of the error.
5.5	The oracle must convert the transaction request and send a web3.js transaction that modifies the state of the blockchain.	Using the oracle, the smart contract can be queried before and after the transaction and it can be seen that the transaction has modified the state of the smart contract.
6.1	The oracle application must be able to subscribe to an event on the blockchain, access the content of the event, and execute instructions after it has detected the event.	An event instantiation can be seen in the source code with a callback that takes the event object as a parameter.
6.2	The oracle application must be able to notify an off-chain application that an event has happened.	An HTTPS POST request containing the data of the event can be seen in the source code of the callback of the event detection.

3.6 Answers to research questions

In this section, we have presented the requirements and design of the system in order to answer **[RQ2] What are the requirements for a model-driven engineering solution for blockchain oracles?** To better answer this question, we have divided it into three sub-research questions.

[RQ2.1] What are the goals of the model-driven engineering solution? The goals can be summarized as creating an intuitive modeling solution that models all the oracles

Table 17. Security requirements and acceptance criteria

ID	Security requirement	Acceptance criteria
1	The MagicDraw plugin must verify that the oracle's certificate path specified in the model is an X509 certificate.	The MagicDraw plugin must read the certificate with the X509 certificate utility of the OpenSSL library and verify that a successful exit code is returned.
2	The MagicDraw plugin must verify that the oracle's private key specified in the model is a valid private key. oracles	The MagicDraw plugin must read the private key with the appropriate key utility of the OpenSSL library and must verify that a successful exit code is returned. The supported key algorithms are RSA and elliptic curve algorithms.
3	The MagicDraw plugin must verify that the private key has a secure size. oracles	The MagicDraw plugin must use the OpenSSL library to read the size of the private key. The size of the key is at least 3072 bits for RSA keys and at least 256 bits for elliptic curve keys.
4	The MagicDraw plugin must verify that the public key in the certificate is the same public key that is derived from the private key. oracles	The MagicDraw plugin must use the OpenSSL library to output the public key from the certificate and from the private key and must verify that the two outputs are identical.
5	The oracle must only accept HTTPS requests. oracles	A client's request to the oracle API should fail if it does not use HTTPS.
6	The oracle must verify that the certificate provided by the client was signed by a trusted certificate authority. oracles	A client's request to the oracle API should fail if the certificate provided is not trusted by the oracle.
7	The oracle must verify that the public key contained in the client's certificate has the required size. oracles	The source code of the oracle server contains a middleware component that verifies the key size of the key in the client's certificate. The requirements for the client's key size are the same as for the server's key size.

identified in Section 2.7 and that can transform the model into a secure standalone oracle application. The solution is designed to be used by users who need to interact with the blockchain, who are not blockchain experts but are familiar with information technologies.

[RQ2.2] What are the use cases of the model-driven engineering solution? The use cases are closely tied to the system goals. The user should be able to model oracles, verify the model and generate code from the model using the MagicDraw plugin. Additionally, the standalone application generated from the model should be able to query smart contracts, update smart contracts' data and subscribe to events from the blockchain. The services of the standalone oracle application should be made accessible to off-chain applications.

[RQ2.3] What are the functional and security requirements of the model-driven engineering solution? The requirements are divided into functional requirements and security requirements. The functional requirements describe the characteristics and required elements of the model, the MagicDraw plugin behavior, and the features of the oracle application generated from the model. The security requirements describe the security features that the plugin and the oracle application should have and in this case specify the characteristics of the X509 certificates used by the oracle application.

4 Implementation

Now that we have defined the requirements for the solution, it is time to implement it. To do this, we first suggest an abstract syntax in the form of a UML class diagram to represent the main concepts of the blockchain oracle domain. The abstract syntax is complemented by OCL constraints which express some restrictions that cannot be shown in the UML model. Then, we suggest a concrete syntax used to model the oracles. The concrete syntax will be used to create the models which will be translated into source code. The concrete syntax is also complemented by OCL constraints. Based on a set of transformation rules, a plugin is developed to read the oracle model and transform it into source code. The architecture of the solution is finally presented using ArchiMate diagrams.

In this section, we answer research question **[RQ3] How to model secure blockchain oracles using model-driven engineering?** To better answer this question, we divide it into three sub-research questions. The first sub-research question is **[RQ3.1] What is the abstract syntax of the model-driven engineering solution?** The second sub-research question is **[RQ3.2] What is the concrete syntax of the model-driven engineering solution?** The third sub-research question is **[RQ3.3] How is the oracle model translated into source code?**

4.1 Blockchain Oracles

This section presents a UML model of oracles. First, the abstract syntax, also known as a meta-model, shows the main concepts of the domain of blockchain oracles and the relationships between these elements. Then, the OCL constraints that accompany the abstract syntax to express restrictions that cannot be shown through the model are described. After that, the concrete syntax is described to show what an actual model of an oracle would look like. Finally, the OCL constraints that accompany the concrete syntax are described.

4.1.1 Abstract Syntax of the Model

The abstract syntax of the proposed model is shown in figure 4. To describe the abstract syntax, the following section first describes the classes and their attributes and then the association classes between the classes.

Classes Three UML classes are used in the abstract syntax to describe the different participants that are involved in an oracle communication process. The *SmartContract* class represents the blockchain application. Its *id* attribute is used to uniquely identify the smart contract and is helpful when defining OCL constraints. The *Bridge* class represents a standalone application generated automatically from the model. This bridge

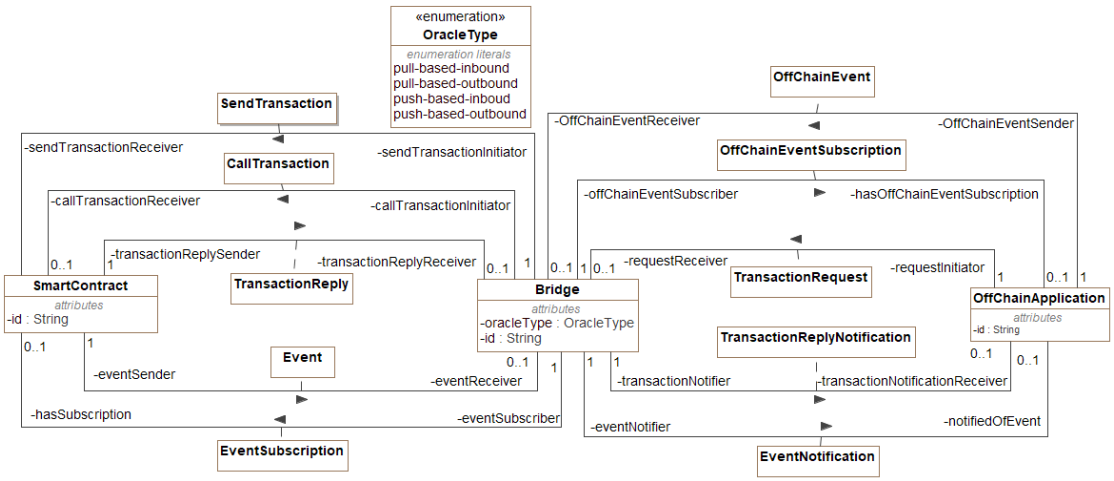

Figure 4. Abstract syntax of oracles domain

acts as a middle man between traditional applications and the Ethereum blockchain and is used to abstract the complexities of making Ethereum transactions and subscribing to Ethereum events. It allows a traditional application to carry out these actions through HTTP requests. The *id* attribute is used to uniquely identify the bridge and is helpful when defining OCL constraints. The *oracleType* is there to show that each model of an oracle should only model one type of oracle at a time. Its value must be one of the values defined in the *OracleType* enumeration. The *OffChainApplication* class represents any off-chain application interacting with the blockchain. For example, it could be an application that needs to store data permanently on the blockchain or which needs to query data from the blockchain. Its *id* attribute is used to uniquely identify it and is helpful when defining OCL constraints.

Association classes The association classes are used to represent communication objects between two different participants (classes). The *TransactionRequest* association class represents a request from an off-chain application to communicate with the blockchain. This association class can represent both a request to modify or query the blockchain. The *SendTransaction* association class denotes a transaction from the *Bridge* to a *SmartContract* that will modify the state of the blockchain by storing additional data on it or modifying data on it. The *CallTransaction* association is similar to the *SendTransaction* association class, but it is used to query the blockchain without modifying its state. The *TransactionReply* association class represents a reply from either a *SendTransaction* or a *CallTransaction*. In the case of the *SendTransaction*, this could be a transaction confirmation for example, and for the *CallTransaction* this would be the data that was requested. The *TransactionReplyNotification* represents the notification to an *OffChainApplication* of a *TransactionReply*. This is basically the same thing as

the *TransactionReply*, but it is between the *Bridge* and the *OffChainApplication*. The *EventSubscription* association class is used to represent the action of a subscription to an *Event* from a *SmartContract*. The *Event* association class is used to represent an event that is emitted by a *SmartContract*. The *EventNotification* association class is used to represent a notification to an off-chain application that an *Event* has been emitted from a *SmartContract*. The *OffChainEventSubscription* is used to represent a subscription to an event that happens off-chain. The *OffChainEvent* is used to represent an event emitted by an *OffChainApplication*.

4.1.2 OCL Constraints of the Abstract Syntax

With the current cardinalities, there are some relationships in the abstract syntax that have no ambiguity. For instance, the abstract syntax clearly shows that there can be at most one *TransactionRequest* associated with the *Bridge*. The same is true for the *EventSubscription* and the *OffChainEventSubscription*. However, it doesn't say that there cannot be both an *EventSubscription* and an *OffChainEventSubscription* even though it is true. The reason is that these two elements belong to two different types of oracles, so they should never be present in the diagram at the same time. This cannot be shown in the diagram, so it must be expressed through OCL constraints. Which elements can be present in the model depends on the type of oracle. The type of oracle is represented by the *oracleType* attribute of the *Bridge*. In all types of oracles, the *Bridge*, the *SmartContract*, and the *OffChainApplication* classes are always present, so there must be OCL constraints for that:

context Bridge **inv**:
self->notEmpty()

context SmartContract **inv**:
self-> notEmpty()

context OffChainApplication **inv**:
self-> notEmpty()

When there is a *Transaction* and a *TransactionReply*, the *Bridge* sending the *Transaction* should be the same as the *Bridge* which receives the reply, and the *SmartContract* receiving the transaction should be the same as the one sending the reply.

context Bridge **inv**:
self.SCTransaction->notEmpty() and self.SCTransactionReply->notEmpty() implies
self.transactionReceiver.id = self.transactionReplySender.id

context SmartContract **inv**:

self.SCTransaction->notEmpty() and self.SCTransactionReply->notEmpty() implies
self.transactionInitiator.id = self.transactionReplyReceiver.id

When there is an *EventSubscription* and an *Event*, the *Bridge* subscribing to the *Event* should be the same as the *Bridge* which receives the *Event*, and the *SmartContract* receiving the *EventSubscription* should be the same as the one sending the *Event*.

context Bridge **inv**:
self.EventSubscription->notEmpty() and self.Event->notEmpty() implies
self.hasSubscription.id = self.eventSender.id

context SmartContract **inv**:
self.EventSubscription->notEmpty() and self.Event -> notEmpty() implies
self.transactionInitiator.id = self.transactionReplyReceiver.id

The requester of a *TransactionRequest* should be the same as the receiver of the *TransactionReplyNotification*. The receiver of the *TransactionRequest* should be the same as the sender of the *TransactionReplyNotification*.

context Bridge **inv**:
self.TransactionRequest->notEmpty() and self.TransactionReplyNotification->notEmpty() implies
self.requestInitiator.id = self.transactionNotificationReceiver.id

context OffChainApplication **inv**:
self.TransactionRequest->notEmpty() and self.TransactionReplyNotification -> notEmpty() implies
self.requestReceiver.id = self.transactionNotifier.id

The receiver of an *OffChainEventSubscription* should be the same as the emitter of the *OffChainEvent*. The owner of a *OffChainEventSubscription* should be the same as the receiver of the *OffChainEvent*.

context Bridge **inv**:
self.OffChainEventSubscription->notEmpty() and self.OffChainEvent->notEmpty() implies
self.hasOffChainEventSubscription.id = self.offChainEventSender.id

context OffChainApplication **inv**:
self.OffChainEventSubscription->notEmpty() and self.OffChainEvent -> notEmpty()

implies
self.offChainEventSubscriber.id = self.offChainEventReceiver.id

Pull-based inbound oracle constraints The pull-based inbound oracle should have an *EventSubscription*, an *Event*, an *EventNotification*, a *TransactionRequest*, and a *Transaction*.

context Bridge **inv**:
self.oracleType = OracleType::pull-based-inbound implies
self.EventSubscription -> notEmpty() and
self.Event -> notEmpty() and
self.EventNotification -> notEmpty() and
self.TransactionRequest -> notEmpty() and
self.SCTransaction -> notEmpty()

There should be no *OffChainEventSubscription* and no *OffChainEvent* in the diagram of the pull-based inbound oracle, so we need OCL constraints for those.

context Bridge **inv**:
self.oracleType = OracleType::pull-based-inbound implies
self.OffChainEventSubscription -> isEmpty()
self.OffChainEvent -> isEmpty()

Pull-based outbound oracle constraints In the pull-based outbound oracle, there should be a *TransactionRequest*, a *SCTransaction*, a *TransactionReply*, a *SCTransaction-Reply*, and a *TransactionReplyNotification*.

context Bridge **inv**:
self.oracleType = OracleType::pull-based-outbound implies
self.TransactionRequest -> notEmpty() and
self.SCTransaction -> notEmpty()
self.TransactionRply -> notEmpty() and
self.TransactionReplyNotification -> notEmpty()

In the pull-based outbound oracle, there should be no *OffChainEventSubscription*, *OffChainEvent*, *EventSubscription* or Event.

context Bridge **inv**:
self.oracleType = OracleType::pull-based-outbound implies
self.OffChainEventSubscription -> isEmpty() and
self.OffChainEvent -> isEmpty() and

self.EventSubscription -> isEmpty() and
self.Event -> isEmpty()

Push-based outbound oracle constraints In the push-based outbound oracle, there should be an *EventSubscription*, an Event, and an *EventNotification*.

context Bridge **inv**:
self.oracleType = OracleType::push-based-outbound implies
self.EventSubscription -> notEmpty() and
self.Event -> notEmpty()
self.EventNotification -> notEmpty()

In the push-based outbound oracle, there should be no *TransactionRequest*, *SCTransaction*, *SCTransactionReply*, *TransactionReplyNotification*, *OffChainEventSubscription*, or *OffChainEvent*.

context Bridge **inv**:
self.oracleType = OracleType::push-based-outbound implies
self.TransactionRequest -> isEmpty() and
self.SCTransaction -> isEmpty() and
self.SCTransactionReply -> isEmpty() and
self.TransactionReplyNotification -> isEmpty() and
self.OffChainEventSubscription -> isEmpty() and
self.OffChainEvent -> isEmpty()

Push-based inbound oracle constraints In the push-based inbound oracle, there should be an *OffChainEventSubscription*, an *OffChainEvent*, and a *SCTransaction*.

context Bridge **inv**:
self.oracleType = OracleType::push-based-inbound implies
self.OffChainEventSubscription -> notEmpty() and
self.OffChainEvent -> notEmpty()
self.SCTransaction -> notEmpty()

In the push-based inbound oracle, there should not be a *TransactionRequest*, a *SCTransactionReply*, a *TransactionReplyNotification*, or an *EventSubscription*, an *Event*, or an *EventNotification*.

context Bridge **inv**:
self.oracleType = OracleType::push-based-inbound implies
self.TransactionRequest -> isEmpty() and

self.SCTransactionReply -> isEmpty() and
self.TransactionReplyNotification -> isEmpty() and
self.EventSubscription -> isEmpty() and
self.Event -> isEmpty() and
self.EventNotification -> isEmpty()

4.1.3 Concrete Syntax of the Model

UML Sequence Diagram The UML sequence diagram is chosen to represent the concrete syntax of the metamodel because it allows the modeling of the main components needed to model oracles: actors, transactions, and events. Also, UML is a well-established modeling solution that comes with an extension mechanism. A lightweight extension of UML is proposed using stereotypes and tagged definitions. Stereotypes allow the extension of native UML elements. In this case, stereotypes are used to extend the UML sequence *lifeline*, which represents the object participating in a transaction in the sequence diagram. For example, a stereotype is created to represent a smart contract, a second stereotype is created to represent a bridge and a third stereotype is created to model an off-chain application. As another example, a stereotype is used in this model to extend the UML sequence *message* to represent a smart contract transaction. Tagged definitions are used to add custom properties to elements of a model. In this case, tagged definitions are used for example to associate a *contractAddress* with a smart contract. Similarly, a tagged definition is used to add a *smartContractFunctionName* attribute to the smart contract transaction to indicate which smart contract transaction is invoked.

Concrete syntax elements This subsection introduces the elements of the concrete syntax. There are 3 lifeline stereotypes, 10 message stereotypes, and 1 combined fragment stereotype. For more information on the stereotypes and tagged values described in this section, the reader can look at the transformation rules section. Further in this section, examples of oracles for each type of oracle show how each of the elements of the concrete syntax is used in practice.

The *SmartContract* lifeline stereotype has 3 tagged values.

- The tagged value *ABI* represents the ABI of the smart contract, which is a JSON data structure describing the functions and events exposed by the smart contract.
- The tagged value *chainID* represents the ID of the network on which the smart contract is hosted. For example, chain ID 1 is the main network of Ethereum, while chain ID 4 represents the Rinkeby test network.
- The *contractAddress* is used to indicate the address of the smart contract which is necessary for the oracle to interact with it.

The *Bridge* lifeline stereotype has 7 tagged values:

- The tagged value *caCertificate* (certificate authority certificate) is the certificate associated with the private key used to sign the certificates that are trusted by the oracle. For instance, when the oracle authenticates the off-chain application, it will verify if the off-chain application's certificate was signed by this certificate.
- The tagged value *serverCertificate* is used to point to the location of the certificate that identifies the oracle. This certificate can be used by the off-chain application to authenticate the oracle.
- The tagged value *serverKey* is the private key associated with the server's certificate. It is used to sign messages sent to the off-chain application and is used by the HTTPS protocol to provide confidentiality and integrity to the communication between the off-chain application and the oracle.
- The *generatedCodeCertificate* tagged value simply states where the code representing the oracle will be generated after the code generating function from the MagicDraw plugin has been executed.
- The *nodeUrl* tagged value represents a blockchain node URL that is used by the oracle to send transactions to the blockchain.
- The *oracleType* tagged value represents the type of oracle that is modeled in the diagram. It is used by the MagicDraw plugin to verify that the required elements for that oracle are present in the diagram and are valid.
- The *port* tagged value represents the port that the oracle listens to for incoming requests.

The *OffChainApplication* lifeline stereotype has 2 tagged values:

- The *hostname* tagged value is the hostname of the off-chain application. It is used by the oracle to know where to send notifications associated with events.
- The *port* tagged value is simply the port associated with the hostname and indicates the port where the off-chain application listens for incoming requests.

The *EventSubscription* message stereotype has 1 tagged value:

- The *eventName* attribute is used to denote what is the name of the event triggered by the associated smart contract.

The *Event* message stereotype does not have any tagged value.

The *EventNotification* message stereotype does not have any tagged value.

The *TransactionRequest* message stereotype does not have any tagged value.

The *SendTransaction* message stereotype has the following tagged values:

- The *gasLimit* tagged value denotes the gas limit that the transaction is allowed to spend.
- The *gasPrice* tagged value is used to indicate how much the initiator of the transaction is willing to pay for the transaction. In the Ethereum network, a higher gas price usually means that the transaction will happen faster.
- The *smartContractFunctionName* tagged value denotes the name of the smart contract function to be called.
- The *privateKey* tagged value denotes the path to the private key to be used to sign that transaction.
- The *numberOfConfirmations* tagged value denotes the number of blocks that must be added to the blockchain before the transaction is considered completed and a confirmation is sent to the off-chain application.
- The *value* tagged value denotes the value in ETH that will be sent by the transaction to the smart contract to which the transaction is made.

The *CallTransaction* message stereotype has 1 tagged value:

- The *smartContractFunctionName* tagged value denotes the name of the smart contract function to be called.

The *TransactionReply* message stereotype has no tagged value.

The *TransactionReplyNotification* has no tagged value.

The *OffChainEventSubscription* combined fragment stereotype has 1 tagged value:

- The *interval* tagged value denotes the interval (in seconds) at which the off-chain application is queried.

The *OffChainQuery* message stereotype has 1 tagged value:

- The *queryPath* tagged value denotes the path at which the off-chain application is queried for data.

The *OffChainQueryReply* message stereotype has no tagged value. Note that the *OffChainQuery* and the *OffChainQueryReply* were not present in the abstract syntax and are part of the OffChainEventSubscription concrete implementation. Also note that for the OffChainEventSubscription, there are elements that are not part of the UML profile. The first element is the combined fragment denoting the condition to trigger the transaction after receiving data from the off-chain application. The second element is the parameter(s) of the transaction.

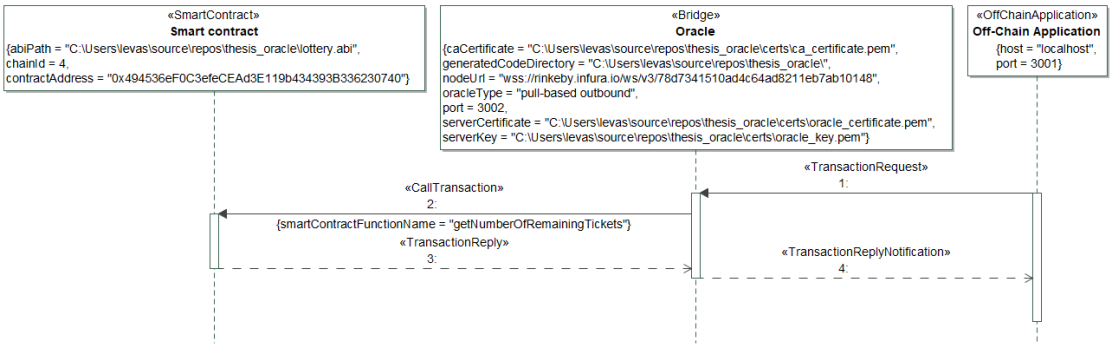

Figure 5. Concrete syntax example of a pull-based outbound oracle

Examples of concrete syntax diagrams Figure 5 shows an example of the concrete syntax used to model a pull-based outbound oracle. The interaction sequence of this oracle starts with a request for data from the off-chain application. That request is translated by the oracle into a call to the smart contract. The smart contract then returns the requested data to the oracle which in turn returns it to the off-chain application.

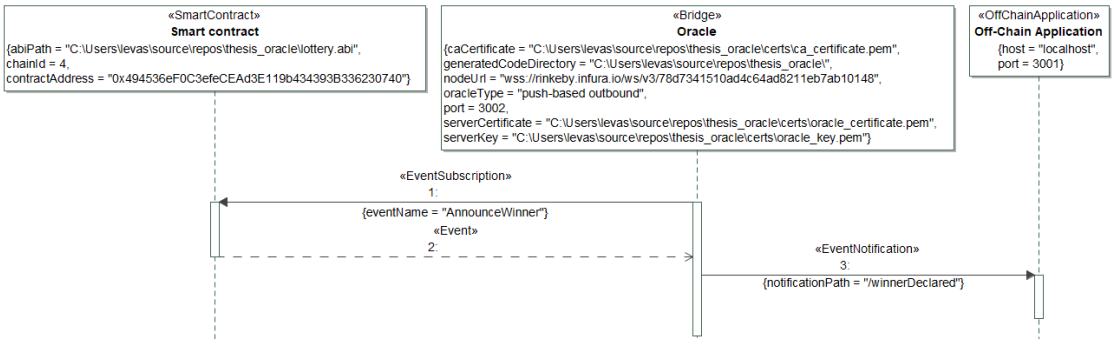

Figure 6. Concrete syntax example of a push-based outbound oracle

Figure 6 shows an example of the concrete syntax used to model a push-based outbound oracle. In this type of oracle, the oracle subscribes to an event of a specific smart contract. When the event is detected, the notification is sent to the off-chain application using the path configured in the notification element.

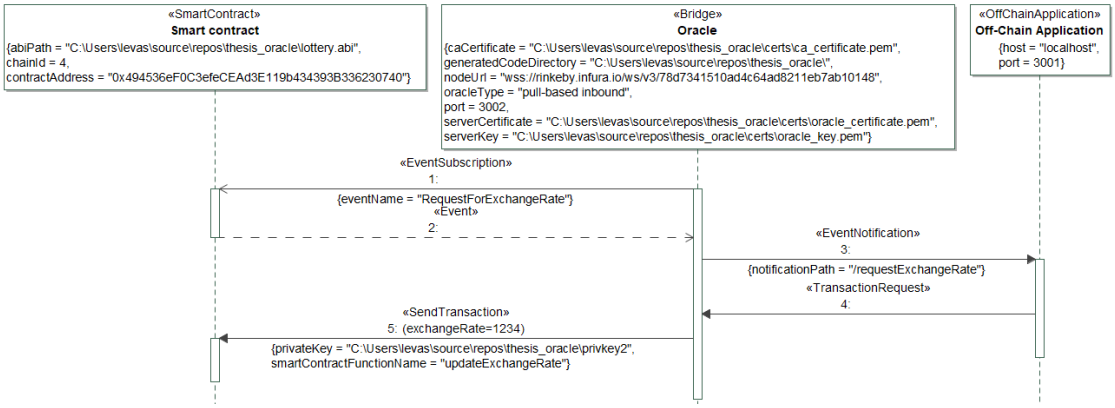

Figure 7. Concrete syntax example of a pull-based inbound oracle

Figure 7 shows an example of the concrete syntax used to model a pull-based inbound oracle. This oracle starts the same way as the push-based outbound. The smart contract emits an event that is picked up by the oracle which notifies the off-chain application. The difference with the push-based outbound oracle is that in this case the event is used to inform the off-chain application that some data is requested. The off-chain application then proceeds to send that data to the oracle which translates the transaction request into a smart contract function call.

Figure 8 shows an example of the concrete syntax used to model a push-based inbound oracle. In this example, an off-chain listener can be configured to poll an off-chain application at a specified interval. Once a certain condition is detected, the modeled transaction is created and sent to blockchain in order to update the smart contract's data. This is used to send data to the blockchain when a certain off-chain event happens.

4.1.4 OCL Constraints of the Concrete Syntax

Some OCL constraints must be applied to the concrete syntax of the model since the concrete syntax has different attributes which cannot be accessed from the abstract syntax. These constraints are enumerated below.

All three tagged values of the *SmartContract* must be defined.

context SmartContract **inv**:
self.abiPath->notEmpty() and

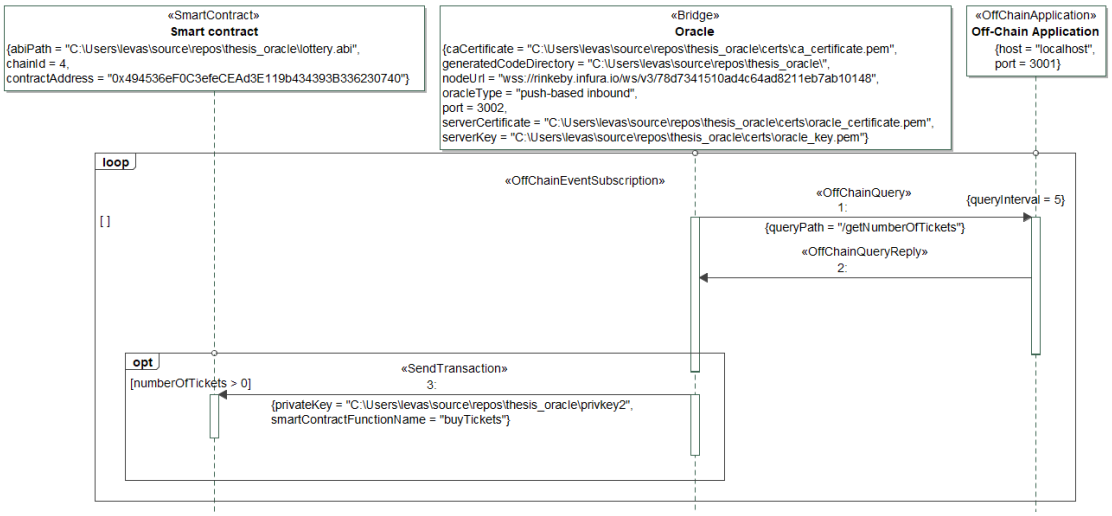

Figure 8. Concrete syntax example of a push-based inbound oracle

self.chainId-> notEmpty() and
self.contractAddress-> notEmpty()

The following tagged values of the *Bridge* must be defined.

context Bridge **inv**:
self.caCertificate->notEmpty() and
self.generatedCodeDirectory-> notEmpty() and
self.nodeUrl-> notEmpty() and
self.oracleType-> notEmpty() and
self.port-> notEmpty() and
self.serverCertificate-> notEmpty() and
self.serverKey-> notEmpty()

The following tagged values of the *OffChainApplication* must be defined.

context OffChainApplication **inv**:
self.host->notEmpty() and
self.port-> notEmpty()

The following tagged value of the *EventSubscription* must be defined.

context EventSubscription **inv**:
self.eventName->notEmpty()

The following tagged value of the *EventNotification* must be defined.

context EventNotification **inv**:
self.notificationPath->notEmpty()

The following tagged value of the *SendTransaction* must be defined.

context SendTransaction **inv**:
self.privateKey->notEmpty()

If the *smartContractFunctionName* attribute is not defined, then the *value* attribute must be defined. That would be the case if someone wants to only send some ETH to another wallet.

context SendTransaction **inv**:
self.smartContractFunctionName-> isEmpty() implies
self.value-> notEmpty()

The following tagged value of the *OffChainEventSubscription* must be defined.

context OffChainEventSubscription **inv**:
self.queryInterval->notEmpty()

The following tagged value of the *OffChainQuery* must be defined.

context OffChainQuery **inv**:
self.queryPath->notEmpty()

4.2 Source Code Generation

This section describes the two main steps of the process of the source code generation from the MagicDraw oracle model. It starts by describing the model verification step and then describes the transformation rules mapping each element of the model to its corresponding element in the source code.

4.2.1 Model Verification

Before the model can be transformed into source code, it must be read into memory. In this thesis, a MagicDraw plugin is used to read the model from the MagicDraw modeling environment. For the model to be properly read by the plugin, first there need to be some rules to determine whether or not the model is valid. Some of these rules are defined in 4.1.2 and are implemented in Java code in the plugin. Other rules are also verified

by the plugin in Java, but they cannot be represented by OCL constraints. For example, all files included in the model should exist. The messages should respect a predefined order for each type of oracle. The parameters associated with the smart contract function to be called should be valid too. The model verification feature is integrated within the source code generation feature, meaning that it is triggered automatically when the user tries to generate the source code. If the model verification fails, then an error message is displayed to indicate why the verification has failed and the source code is not generated until the model is rectified.

4.2.2 Transformation Rules

The remaining of this section describes how the elements of the concrete syntax of the model (the stereotypes and tagged values) are used to generate the source code of the oracle application.

TR1: SmartContract stereotype and tagged values The tagged value *abiPath* of the *SmartContract* stereotype is used to indicate the location of the contract's ABI. This attribute is used to read a JSON object representing the smart contract into memory. This JSON object is then used to create a *web3.js Contract* instance. This instance will be needed by the *EventSubscription* which is described further below. The *chainId* tagged value is one of the required parameters when creating a *web3.js* transaction, so it is added as a parameter to each *SendTransaction* element that is associated with that *SmartContract*. The *contractAddress* tagged value is used in a similar fashion to *chainId*. It is also a required parameter for the *SendTransaction* element and is added to the attributes of that source code element.

TR2: Bridge stereotype and tagged values The *port* is the port at which the Node.js application listens for requests. It is used as a parameter in the constructor of the Node.js server. The *caCertificate* tagged value is used as a constructor parameter when creating the Node.js server. It is used to authenticate the requests from the off-chain application. When an off-chain application makes a request, it offers a certificate as proof of identity. The oracle verifies that identity by making sure that the offered certificate has been signed by the *caCertificate*. The *serverCertificate* and *serverKey* are both used as HTTPS request parameters to allow the receiving party to authenticate the *Bridge*. They are used in the HTTPS request that is represented by the *EventNotification* and the *OffChainQuery* in the model. The *generatedCodeDirectory* tagged value is the output path of the code generated by the plugin. The *nodeUrl* is used as a parameter of the constructor of the *WebsocketProvider*, which is the component through which the *Bridge* can interact with the blockchain through an Ethereum node. The *oracleType* is not used to generate the source code. It is there to force the user to consciously make a choice and it makes it

easier to verify the model because the type of oracle in the diagram does not have to be guessed.

TR3: OffChainApplication stereotype and tagged values The *host* and *port* are both used when making HTTPS requests to an *OffChainApplication*. They are used as parameters of the HTTPS requests represented by the *EventNotification* and *OffChainQuery*. For example, an *OffChainQuery* will access the *host* and *port* attributes of the *OffChainApplication* to which it is pointing.

TR4: EventSubscription stereotype and tagged values The *eventName* tagged value is used as a key to the *events* dictionary of the *web3.js* contract object. This allows accessing the desired event object from which it is possible to instantiate the event subscription.

TR5: EventNotification stereotype and tagged values The *EventNotification* stereotype is mapped to a callback function on the event subscription function. This means that when an event is detected, then the *EventNotification* callback function is called, which creates an HTTPS request to the *OffChainApplication* to notify that an *Event* has happened. The *notificationPath* tagged value is used as the *path* parameter in the event notification HTTPS request.

TR6: SendTransaction stereotype and tagged values The *numberOfConfirmations* and *confirmationPath* tagged values are optional. If they are defined, then a callback function producing an HTTPS POST request will be generated in the source code, along with the code evaluating the number of blocks that were added since the transaction was included in a block. This POST request will act as a confirmation and it will be sent at the specified path (URL) after the specified number of confirmations has been received. The *gasPrice* and *gasLimit* tagged values are also optional. They are used to specify the gas price and the gas limit parameters of the *web3.js* transaction object. If they are not specified, then some default values are used. The default value of the gas price is the "high" value returned from Eth Gas Station (`https://ethgasstation.info/json/ethgasAPI.json`). It is mentioned on this site that "high" gasPrice means that the transaction will be included in a block faster than 90% of other submitted transactions. The default gas limit is 100,000 since the same project (Eth Gas Prices), it is mentioned that most simple transfers use an average of 21,000 Gwei, so 1,000,000 Gwei is used here because it is much higher than the simple transaction average which means that most transactions will not run out of gas using that limit. The *smartContractFunctionName* and the *privateKey* tagged values are additional parameters of the *web3.js* transaction object. Respectively, they specify the smart contract function name that will be called by the transaction, and the private key used to sign the transaction. The *value* tagged value

is another parameter of the *web3.js* transaction object and is used to indicate how much ETH will be sent by the transaction. The *smartContractFunctionName* tagged value can be omitted if the *value* tagged value is defined. In that case, the default payable function of the smart contract will be called.

TR7: OffChainEventSubscription stereotype and tagged values When the OffChainEventSubscription stereotype is present in the model, then a callback method will be created to query an off-chain application at a specified interval. The *queryInterval* tagged value specifies the interval in seconds after which the callback is executed.

TR8: OffChainQuery stereotype and tagged values The *queryPath* tagged value will be used as the path of the HTTPS GET request generated in the previous transformation rule.

Model elements with no association rules There are some elements that must be present in the model for clarity and to allow model verification, but that are not transformed into source code. These elements are described below. The *TransactionRequest* Message stereotype is used to show an interaction initiated from the *OffChainApplication*, so it is out of the scope of the elements generated from the model. The *CallTransaction* Message stereotype is used to show that the *Bridge* will query the blockchain for an *OffChainApplication* following a *TransactionRequest*. However, the details of the *SmartContract* function to call must be provided dynamically in the *TransactionRequest* by the *OffChainApplication*, so this cannot be generated from the model. The *TransactionReply* Message stereotype is used to show the reply of a *CallTransaction* or a *SendTransaction* and is generated by the *SmartContract*, which means it cannot be generated from the model. Similarly, the *Event* stereotype is generated by the *SmartContract* and does not result in generated code for analog reasons. Finally, the *OffChainQueryReply* Message stereotype is used to show the reply of the *OffChainQuery* and is created dynamically by the *OffChainApplication* so it cannot be generated from the model.

4.3 System Architecture

The objective of this section is to give an overview of the architecture of the MagicDraw plugin and the generated Oracle by enumerating the most important components. First, the project dependencies are briefly described. Then, the system components are represented in Archimate diagrams.

4.3.1 System Dependencies

MagicDraw MagicDraw is a visual modeling tool that supports many modeling languages such as UML. It is particularly interesting for this thesis because it provides an API that can be used to interact programmatically with the application and it allows developers to create plugins that can be integrated into the application to add new functionalities. The MagicDraw API is a set of Java interfaces and classes that provide the developer with access to functionalities. For example, the API can be used to add elements such as buttons to the graphical interface of the application and add events to these buttons. It also makes it possible for the developer to read diagrams created with MagicDraw and to modify or add new elements. One important class of MagicDraw is the *Plugin* class. By extending this class, the developer effectively creates a class that represents a new plugin. This class and the other classes that make up the plugin then need to be compiled and placed in a specific folder in the MagicDraw installation directory, after which MagicDraw will automatically load the plugin when it is started. The developer can also find some working examples of plugins in the installation directory of MagicDraw, as well as the documentation for using the API. Finally, it is possible to run MagicDraw from an IDE like Intellij or Eclipse in order to debug the code dynamically and make modifications to the plugins without having to close and reopen the whole application.

Node.js Nodes.js is an open-source runtime environment that is typically used to write web servers in Javascript. It is quick to set up and countless examples can be found online to get started. It is also easy to configure a Node.js server to use SSL certificates for authenticating clients and to ensure that all communication to and from the server is encrypted.

OpenSSL The OpenSSL library is an open source library that provides cryptographic functions and utilities. In this thesis, it is used by the MagicDraw plugin to read the attributes of X509 certificates and private keys.

web3.js web3.js is a set of libraries that can be used to interact with a blockchain node to indirectly communicate with a blockchain.

One important aspect of web3.js is the provider. The provider is the communication method with which a client can communicate with a node. For example, HTTP, Websocket, or IPC can be used. In section System Components, the *nodeUrl* tagged value of the *Bridge* is used to create a WebSocket connection to a blockchain node. It is important to note that a provider only allows communication with one chain at a time. To communicate with another chain, a new provider must be created.

Another important element of the web3.js libraries is the *Contract* object. This object is created using the ABI of a smart contract and its address. The ABI of the

contract is a JSON object that describes the functions and the events of the contract. The *Contract* object is useful in two ways for this thesis. The *Contract* object is first useful for accessing the events of the smart contract. Once the object is created, it has an attribute called *events* which is a dictionary containing all events of the smart contract. When an oracle needs to trigger an action after an event, it calls a function on the event and defines a callback function to be run when the event is triggered. The *Contract* object is also useful for accessing the methods of the smart contract. As for the events, the contract object has an attribute called *methods* which is a dictionary containing the methods of the smart contract. This attribute is used by the oracle to call smart contract functions, which can be used to query the blockchain or modify the state of the smart contract.

4.3.2 System Components

Two models have been created to represent the system architecture and both of them have been made using the ArchiMate modeling tool. In this model, the upper yellow layer represents the services that each system component offers. For instance, one of the services offered by the *MagicDraw Modeling Application* is the ability to create a model. This list is not exhaustive and the elements which are present are the elements that are relevant to this thesis. For example, MagicDraw provides other services that are not enumerated in these diagrams. The blue layer is used to represent the main application software components that make the realization of the yellow layer possible. Finally, the green layer is used to represent the inputs and outputs of the blue layer [Arc].

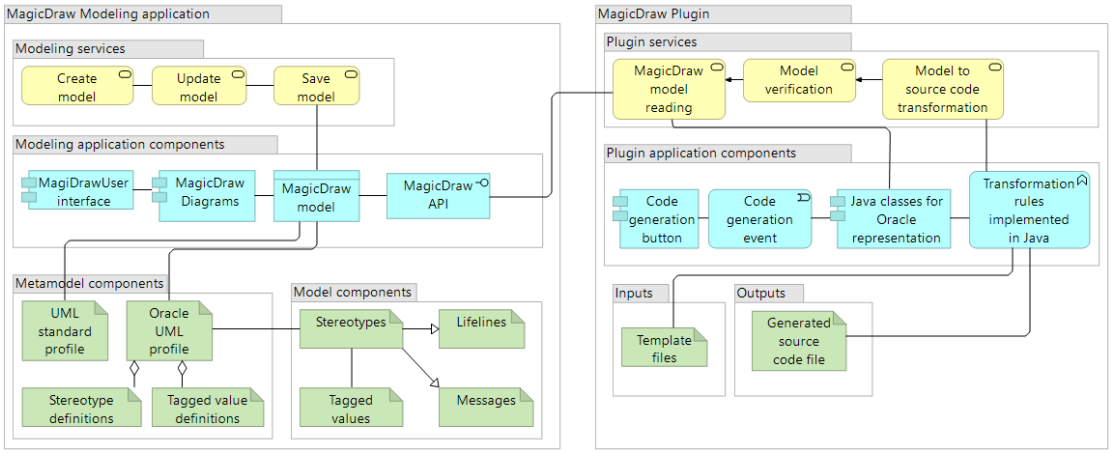

Figure 9. MagicDraw Plugin architecture diagram

The services provided by the MagicDraw modeling application that are relevant for the contribution of this thesis are the services related to the creation of a model, namely the *Create model*, *Update model*, and *Save model* services. The software components that support these services are the user interface, the diagrams of the model, and the

MagicDraw API. The MagicDraw API provides a number of functions to interact with the MagicDraw application. In this case, only the functions used to read the diagrams and the model are used. In order to create these models, the UML standard profile containing the definition of UML elements and the Oracle UML Profile containing the stereotypes and tagged values necessary to represent the oracle components are needed.

The MagicDraw plugin developed in this thesis depends on the MagicDraw model created using the MagicDraw modeling application. The services offered by the plugin are the reading of the MagicDraw model and diagrams, the verification of the model to make sure that the modeled oracles respect the rules defined by the abstract and concrete syntax, and the transformation of the model to the source code. The software components that support the MagicDraw plugin services are a MagicDraw user interface button that emits a MagicDraw event that will be received by the plugin to start the code generation process. Another software component is the set of Java classes that are used to represent the different elements of the model. Finally, the last software component is the set of transformation rules that transform the Java objects that represent the oracles into a standalone oracle application.

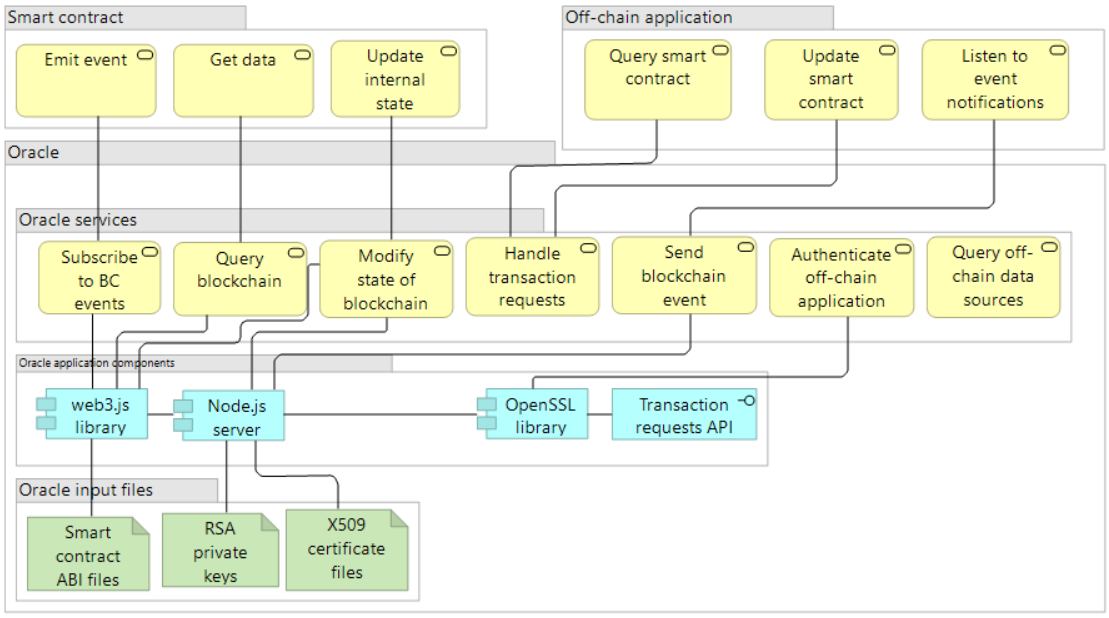

Figure 10. Oracle application architecture diagram

Figure 10 represents the architecture of the standalone application which is the result of the code generation process. Note that the Smart contact components as well as the Off-chain application components shown in this figure are not generated by the MagicDraw plugin and are only shown to depict the interactions with the oracle components. This shows that the oracle acts as a middleman between off-chain applications and smart

contracts, which both want to receive and send data.

The oracle provides many services. Some of these services must be configured from the oracle model. That is the case for the blockchain events subscriptions, the blockchain event notifications, the authentication of the off-chain application, and the queries made to off-chain applications. Other services can be called dynamically without being configured in the model. That is the case for querying the blockchain and modifying the state of the blockchain through transaction requests.

There are 4 software components that support the services of the oracle. The *web3.js* javascript library is used to communicate with the blockchain through HTTP requests sent to a blockchain node. The blockchain node used in this project is an Infura node. Anyone can have access to an Infura node by simply creating an account on their website (`https://infura.io/`). The *web3.js* library is the component that makes it possible for the oracle to query and update the blockchain, as well as to subscribe to blockchain events. The *Node.js* server is what makes the oracle available to the off-chain applications by listening for requests on a specific port. The *OpenSSL* library is used by the *Node.js* server to verify the X509 certificates offered by the clients (off-chain applications) which make requests to the oracle. Finally, the *Transaction requests API* receives transactions from off-chain applications to either query or modify the state of the blockchain and translates these HTTPS requests into blockchain transactions using the *web3.js* library.

4.4 Answers to Research Questions

In this section, the contribution of this thesis was described in order to answer **[RQ3] How to model secure blockchain oracles using model-driven engineering?** To answer this question more specifically, we need to address each subquestion individually.

[RQ3.1] What is the abstract syntax of the model-driven engineering solution? An abstract syntax is used to represent the high-level concepts of the oracle domain. The abstract syntax is represented using a UML class diagram. Then, the abstract syntax constraints are expressed using OCL constraints to provide additional details that cannot be represented in the class diagram. **[RQ3.2] What is the concrete syntax of the model-driven engineering solution?** The concrete syntax is suggested to model the oracles that will be transformed into source code. For the concrete syntax, the UML sequence diagram was chosen, because it is an interaction diagram that contains the main elements required for a blockchain oracle. The necessary elements to model oracles are actors, ordered messages, and events which are all available in the UML sequence diagram. The elements of the concrete syntax are organized into a UML profile which contains stereotypes and tagged values. Finally, OCL constraints are defined on the concrete syntax to complete the constraints of the abstract syntax by accessing the attributes that are only available in the concrete syntax. **[RQ3.3] How is the oracle model translated into source code?** MagicDraw was chosen as the modeling tool in this thesis. Before translating the model into source code, the model must be read. A MagicDraw plugin

is developed to read the model in memory and verify it. This plugin makes use of the MagicDraw developer API which provides functions to access elements of diagrams of the MagicDraw modeling application. Some transformation rules are defined to map the elements of the concrete syntax to elements of the source code. The MagicDraw plugin implements these rules and writes the resulting source code to a file.

5 Evaluation

Now that the contribution has been implemented, it must be evaluated to confirm that it meets the requirements based on the acceptance criteria. To achieve this, an evaluation scenario is designed with the objective of providing an opportunity to verify all of the functional and security requirements and their associated acceptance criteria. The main research question of this section is **[RQ4] What is the usability of the proposed solution for MDE of blockchain oracles?** To better answer this question, we have divided it into three sub-research questions. The first sub-research question is **[RQ4.1] What would be a proper evaluation scenario for the contribution?** The second sub-research question is **[RQ4.2] Does the solution satisfy the functional requirements?** The third sub-research question is **[RQ4.3] Does the solution satisfy the security requirements?**

5.1 Evaluation Scenario

The scenario chosen for the evaluation of the contribution of this thesis is a lottery smart contract. A smart contract containing source code that describes a lottery is published on the Ethereum blockchain. The advantage of using the blockchain for this type of application is that the source code can be shared to the public and potential participants in the lottery can verify that the smart contract represents a legitimate lottery application. After the smart contract is published on the blockchain, the participants can have the guarantee that the smart contract will behave exactly according to its instructions.

The lottery works as follows. Anyone can participate by sending a multiple of 0.001 ETH to the smart contract. For each 0.001 ETH, the participant gets one lottery "ticket" that represents one entry in the lottery to win a fixed and predetermined amount in USD. Once the required number of tickets has been reached, the smart contract then calls an oracle to get the amount of ETH that is equivalent to the lottery prize in USD. Then, the lottery winner is randomly determined and the prize amount in ETH is then sent to the winner and the lottery ends.

For evaluation purposes, a simple off-chain application has been developed. This application aims to represent an application that would make use of an oracle in real life. It is developed in *node.js* and is composed of a few HTTPS endpoints. The off-chain application has an endpoint called */getNumberOfTickets* which returns the number of tickets that the off-chain application wants to buy. This is used by the push-based inbound oracle described in the following section to trigger a transaction. The off-chain application also has a */setNumberOfTickets* endpoint. This is used to update the number of tickets that the off-chain application will return when the */getNumberOfTickets* endpoint is queried. Another endpoint is */requestExchangeRate*, which is used by the pull-based inbound oracle described in the following section. When queried, this endpoint returns the number of Wei for 1 USD i.e. the exchange rate. Finally, the */winnerDeclared* is used by the push-based outbound oracle to notify the off-chain application that a lottery winner

has been declared. This evaluation scenario was chosen for a few reasons. First, it is an example of an application where using blockchain comes with interesting advantages. More specifically, users have a good reason for wanting data immutability and guaranteed code execution in this scenario, since money is at stake. Additionally, this scenario is relatively simple but makes use of the four types of oracles which the contribution of this thesis aims at generating from the model. This makes it possible to evaluate all of the functional and security requirements.

5.2 Oracle Modeling

This section describes how the four types of oracles described in this thesis are used in the evaluation scenario, namely the push-based inbound oracle, the pull-based outbound oracle, the pull-based inbound oracle, and the push-based outbound oracle.

5.2.1 Push-Based Inbound Oracle

The push-based inbound oracle allows an off-chain application to buy lottery tickets. The oracle creates an off-chain event subscription. The oracle subscribes to an event from the off-chain application which contains the number of lottery tickets to buy. The way this is implemented is that the oracle will make requests to the off-chain application at a specified interval and asks for the number of tickets that the off-chain application wants to buy. When the off-chain application returns a number larger than 0, then the oracle makes the transaction to buy the specified number of tickets according to the *SendTransaction* of the model. Figure 11 shows the model that was used to generate the source code for this oracle.

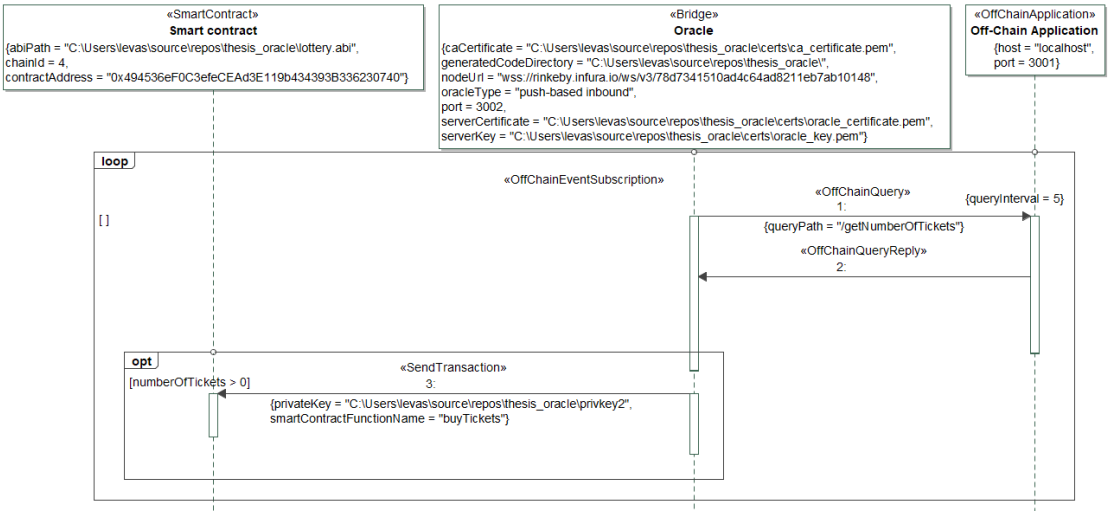

Figure 11. Model of the oracle used to buy lottery tickets

5.2.2 Pull-Based Outbound Oracle

The pull-based outbound oracle is used by the off-chain application to query the smart contract to get the number of tickets remaining before the winner of the lottery is announced. An authenticated HTTPS request is made to the oracle specifying the name of the smart contract view function to call. Figure 12 shows the model that was used to generate the source code for this oracle.

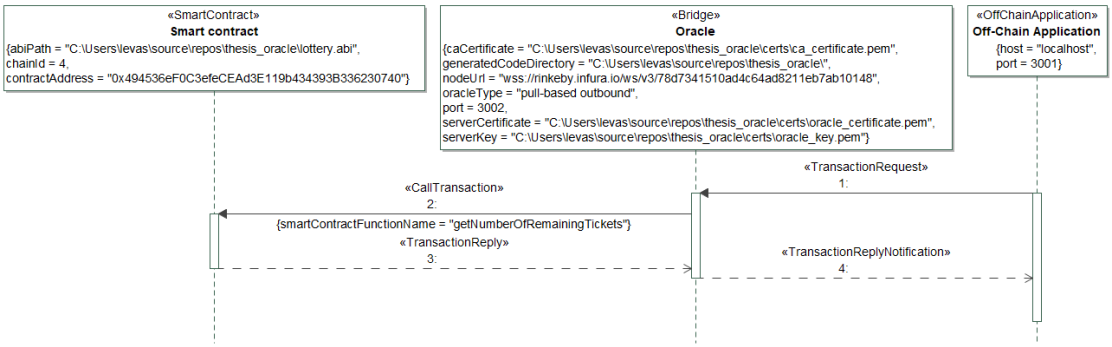

Figure 12. Model of the oracle for getting the number of remaining tickets for the lottery

5.2.3 Pull-Based Inbound Oracle

The pull-based inbound oracle is used by the smart contract when the required number of lottery tickets has been reached. It is used to get the exchange rate in ETH for the lottery prize amount in USD in order to send the right amount of ETH to the winner of the lottery. Only a specified smart contract address is allowed to send this information to the smart contract to reduce the chance of someone sending malicious information. Figure 13 shows the model that was used to generate the source code for this oracle.

5.2.4 Push-Based Outbound Oracle

The push-based outbound oracle is used by the smart contract to announce the winner of the lottery. The event from the smart contract is captured by the oracle which then sends the notification to the appropriate off-chain application. Figure 14 shows the model that was used to generate the source code for this oracle.

5.3 Evaluation of Functional Requirements

In this section, we evaluate the functional requirements section using their corresponding evaluation criteria. As the functional requirements are grouped according to their associated use case, we start by evaluating the functional requirements for the oracle

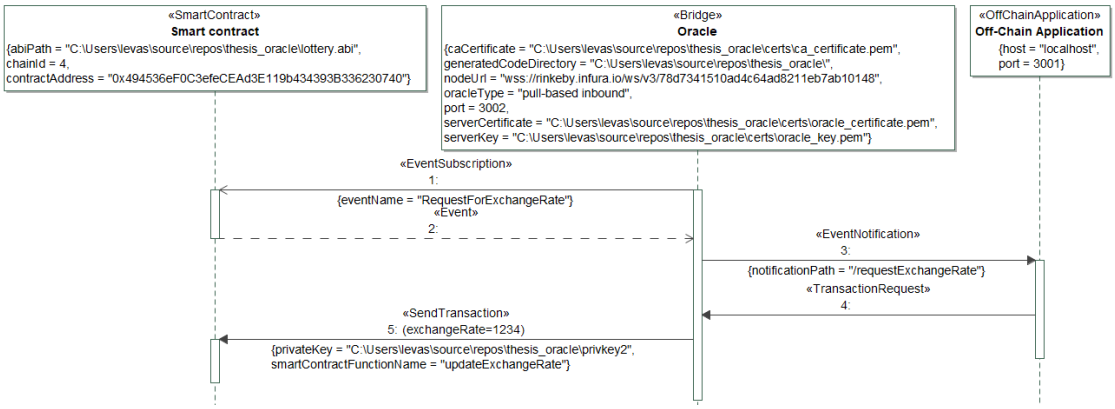

Figure 13. Model of the oracle used by the smart contract to get the exchange rate for the lottery prize

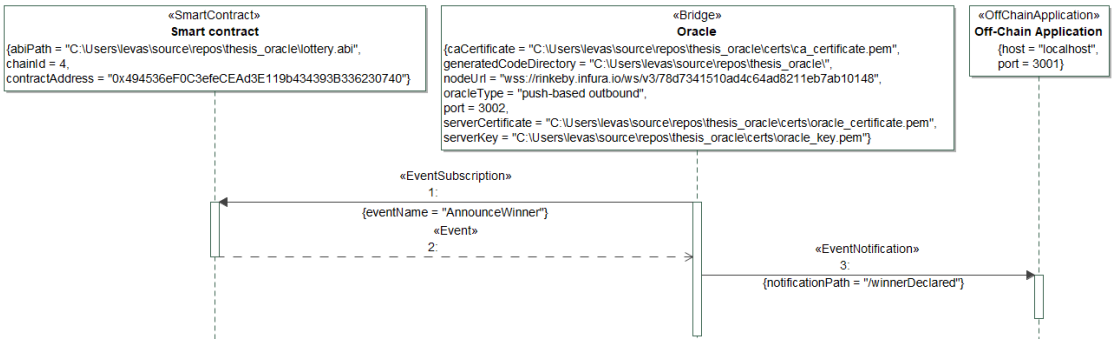

Figure 14. Model of the oracle used to listen to the event announcing the winner of the lottery

modeling use case. Then, we evaluate the model verification, code generation, smart contract query, smart contract update, and event subscription use cases.

5.3.1 Oracle Modeling

In this section, we evaluate the functional requirements related to the modeling of oracles. The associated use case is described in Table 7 and the functional requirements are described in Table 16.

Functional requirement 1.1

Acceptance criteria: The UML Profile can be opened using the MagicDraw application.

Evaluation details: The *OracleProfile* was successfully opened in MagicDraw, as shown

in Figure 15.

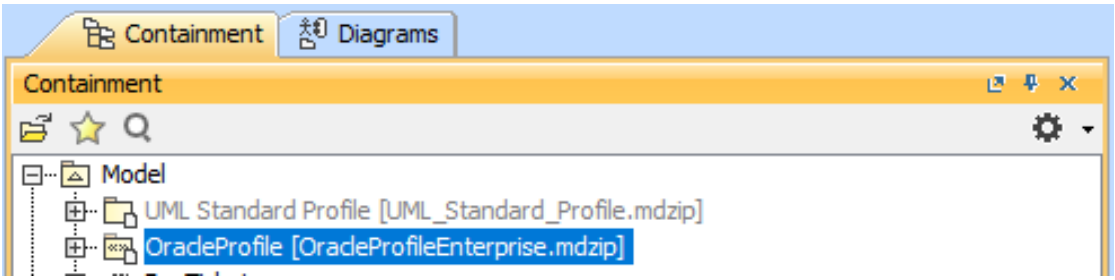

Figure 15. Oracle Profile loaded in MagicDraw

Validation result: Pass

Functional requirement 1.2

Acceptance criteria: The lifeline stereotypes for the actors are shown by expanding the UML Profile in MagicDraw.

Evaluation details: Looking at Figure 16, we can see that the lifeline stereotypes for the actors are the ones where "Lifeline" is written between square brackets.

Validation result: Pass

Functional requirement 1.3

Acceptance criteria: The tag definitions for the actors are shown by double-clicking on an actor lifeline stereotype in MagicDraw.

Evaluation details: Looking at Figure 17, we can see an example of the tagged values for the *SmartContract* lifeline stereotype.

Validation result: Pass

Functional requirement 1.4

Acceptance criteria: The lifeline stereotypes for the messages are shown by expanding the UML Profile in MagicDraw.

Evaluation details: As with the actors' stereotypes, some of the message stereotypes can be seen in Figure 16.

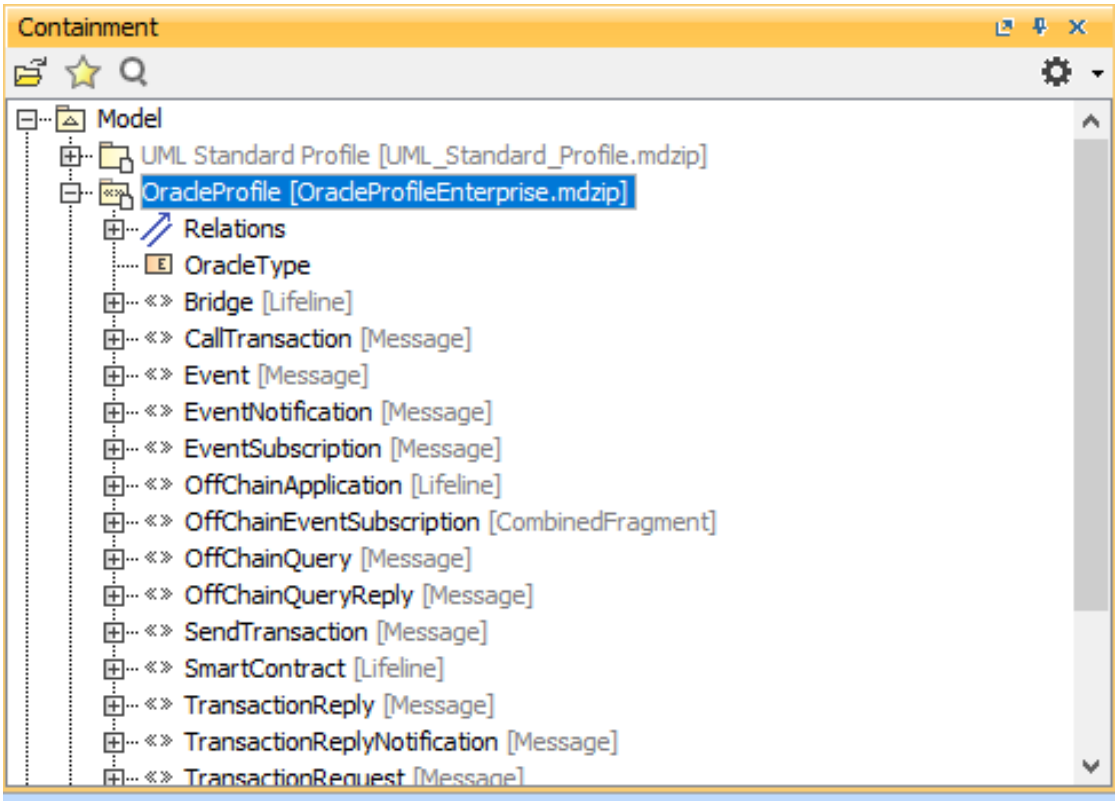

Figure 16. Oracle Profile stereotypes in MagicDraw

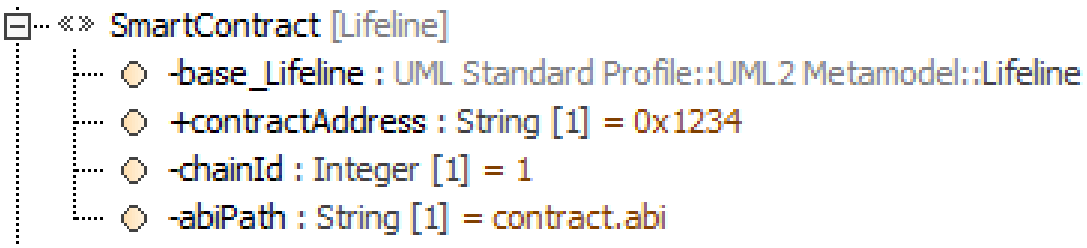

Figure 17. SmartContract tagged values in MagicDraw

Validation result: Pass

Functional requirement 1.5

Acceptance criteria: The tag definitions for the messages are shown by double-clicking on a message lifeline stereotype in MagicDraw.

Evaluation details: Figure 18 shows the tagged value for the *EventSubscription* message stereotype.

Figure 18. EventSubscription tagged values in MagicDraw

Validation result: Pass

5.3.2 Verify Model

In this section, we evaluate the functional requirements related to the verification of the model of oracles. The associated use case is described in Table 8 and the functional requirements are described in Table 16.

Functional requirement 2.1

Acceptance criteria: MagicDraw API function calls can be seen by looking at the plugin's source code.

Evaluation details: By looking at Figure 19, we can see that the *Element*, *Stereotype*, and *StereotypesHelper* classes are used, which are classes of the MagicDraw API. The function shown in this figure takes an element of a diagram along with its stereotype and extracts the tagged values of the element.

Validation result: Pass

Functional requirement 2.2

Acceptance criteria: The MagicDraw plugin shows an error message when the user inputs an invalid oracle type or when the oracle type is missing.

Evaluation details: We can see in Figure 20 an example of the error message that the plugin shows if the oracle type is invalid.

Validation result: Pass

```
private List<Tuple> getTaggedValues(Element element, Stereotype
    stereotype) {
    List<Tuple> result = new ArrayList<>();
    List<Property> attributes = stereotype.getOwnedAttribute();
    for (Property p: attributes) {
        if (p.getName().startsWith("base_")) continue;
        String name = p.getName();
        List values = StereotypesHelper.getStereotypePropertyValue(
            element, stereotype, name);
        if (values.size() > 0) {
            result.add(new Tuple(name, values.get(0)));
        }
    }
    return result;
}
```

Figure 19. Example of MagicDraw API calls

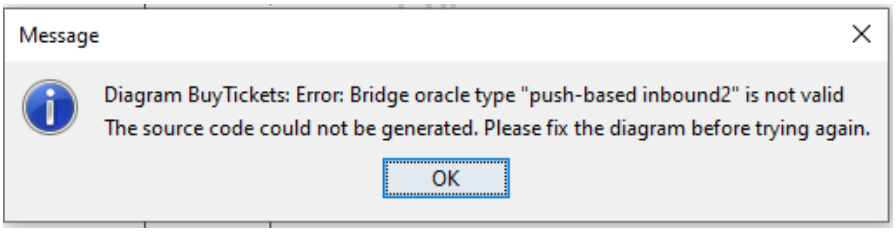

Figure 20. MagicDraw plugin error message: wrong oracle type

Functional requirement 2.3

Acceptance criteria: The MagicDraw plugin shows an error message if a stereotype of the model is missing, invalid, or in the wrong order. The error message indicates which element is missing or invalid and the reason why it is invalid.

Evaluation details: As an example, Figure 21 shows a diagram of a push-based outbound oracle where the last message of the interaction is of the wrong type. The plugin shows an error message indicating which type the last message should be.

Validation result: Pass

Functional requirement 2.4

Acceptance criteria: The MagicDraw plugin shows an error message if a required tagged value is not present. The error message contains the name of the invalid tagged value and

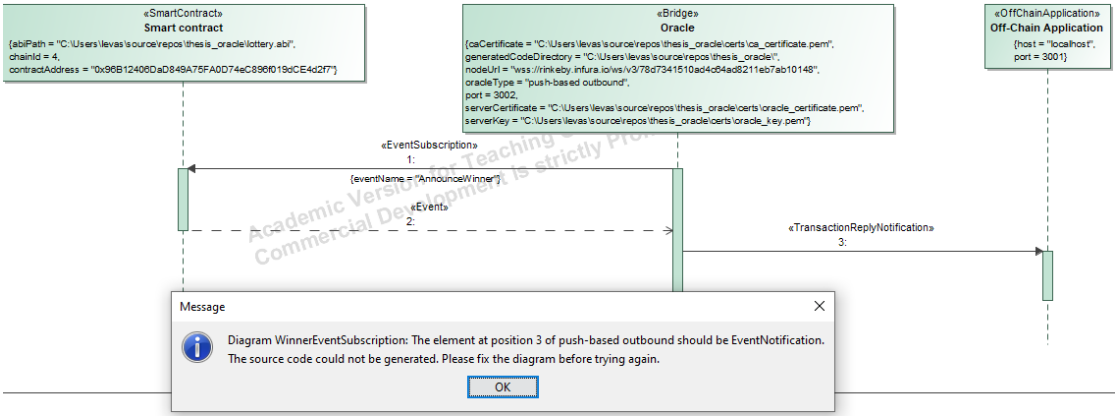

Figure 21. MagicDraw plugin error message: wrong stereotype

the reason why it is invalid.

Evaluation details: For example, Figure 22 shows the error message shown by the plugin when the *EventNotification* stereotype does not have a *notificationPath* tagged value.

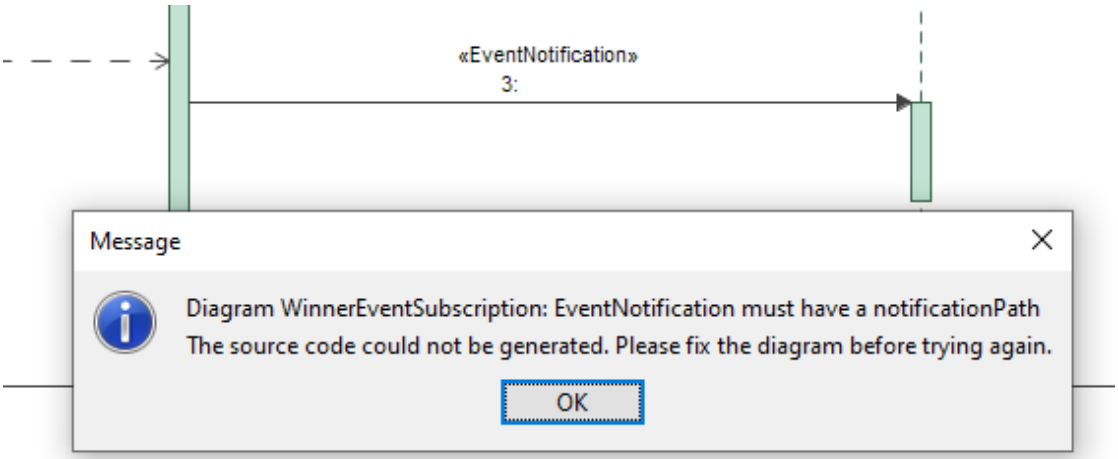

Figure 22. MagicDraw plugin error message: missing tagged value

Validation result: Pass

5.3.3 Generate Code from Model

In this section, we evaluate the functional requirements related to automatic code generation from the model of oracles. The associated use case is described in Table 9 and the functional requirements are described in Table 16.

Functional requirement 3.1

Acceptance criteria: The mapping between elements of the model and source code elements is visible in the source code of the plugin.

Evaluation details: Figure 23 shows the mapping between the model element and the oracle source code for the push-based inbound oracle. A Java String is used as a template for the code of the oracle, and the tagged values from the different diagram elements read by the plugin are injected into the template.

```
ocesString += String.format(OCES_TEMPLATE,
    dataSource.getHost(), // 1
    dataSource.getPort(), // 2
    offChainQuery.getQueryPath(), // 3
    constraint.getVariableName(), // 4
    constraint.getOperator(), // 5
    operandString, // 6
    parametersString, // 7
    transactionConfirmationCode, // 8
    transactionAttributes // 9
);
```

Figure 23. MagicDraw plugin code snippet: mapping between model elements and oracle source code elements

Validation result: Pass

Functional requirement 3.2

Acceptance criteria: A single source code file named oracle.js is found at the path specified in the model after the code generation.

Evaluation details: Figure 24 shows an extract of the source code of the plugin that writes the result of the code generation to a file. We can see that the result is written to a single file and the name of the file is oracle.js.

```
    } catch (InvalidModelException e) {
        String message = e.getMessage();
        String newMessage = "Diagram " + ((Interaction) interaction).
            getName() + ": " + message;
        throw new InvalidModelException(newMessage);
    }
}
String fileContent = OracleCodeGenerator.generateOracleCode(model);
String separator = System.getProperty("file.separator");
String generatedPath = model.getGeneratedCodeDirectory() + separator
   + "oracle.js";
createFile(fileContent, generatedPath);
return generatedPath;
```

Figure 24. MagicDraw plugin code snippet: code generation output

Validation result: Pass

Functional requirement 3.3

Acceptance criteria: There is no source code file generated at the specified directory after an error message is shown by the MagicDraw plugin. If there already was a source code file, then it remains unmodified.

Evaluation details: Figure 24 shows that the code generation result will not be written to the file if the model is invalid since an exception will be thrown.

Validation result: Pass

5.3.4 Query Smart Contract

In this section, we evaluate the functional requirements related to the smart contract query functionality of the automatically generated oracle application. The associated use case is described in Table 10 and the functional requirements are described in Table 16.

Functional requirement 4.1

Acceptance criteria: A response is received by an application making a request to the oracle's API endpoint for blockchain queries.

Evaluation details: Figure 25 shows an example of a curl request used to query the

smart contract function *getNumberOfRemainingTickets* through the oracle. The figure shows that the oracle has managed to fetch the requested data and return it to the client. In this case, the data is returned through the *value* parameter of the JSON response.

```
$ curl \
> https://DESKTOP-FQK0S45.local:3002/callViewFunction \
> --cacert ./ca_certificate.pem \
> --cert offchain_certificate.pem \
> --key offchain_key.pem \
> -d @- <<EOF
>
> {
>     "functionName": "getNumberOfRemainingTickets",
>     "contractAddress": "0x494536eF0C3efeCEAd3E119b434393B336230740",
>     "chainId": 4
> }
> EOF
{"status":200,"value":"0","functionName":"getNumberOfRemainingTickets"}
```

Figure 25. Curl request to query smart contract through oracle

Validation result: Pass

Functional requirement 4.2

Acceptance criteria: The mapping between an API request's parameters and the smart contract function call created by the oracle can be seen in the source code of the oracle.

Evaluation details: We can see in Figure 26 that a call is made to a function called *getViewFunctionRequestArguments* that returns an array of arguments to be passed to the smart contract function.

Validation result: Pass

Functional requirement 4.3

Acceptance criteria: A response with HTTP status 400 is returned to the client when the client makes a request with invalid parameters. The response also contains a message field with a description of the error.

```
const args = getViewFunctionRequestArguments(req.body,
    contractAddress, req.body.functionName);
if (web3Contracts[contractAddress] == undefined) return res.status
    (400).json({status:400, message: 'Unknown contract'}).end();
  web3Contracts[contractAddress].methods[req.body.functionName](...
      args).call().then((result) => {
  return res.status(200).json({status: 200, value: result,
      functionName: req.body.functionName}).end();
});
```

Figure 26. View function call with web3.js

Evaluation details: Figure 27 shows an example of a request to the oracle with a missing parameter. As expected, the oracle returns a 400 status code with a message indicating that the required *chainId* parameter is missing.

```
$ curl \
> https://DESKTOP-FQK0S45.local:3002/callViewFunction \
> --cacert ./ca_certificate.pem \
> --cert offchain_certificate.pem \
> --key offchain_key.pem \
> -d @- <<EOF
>
> {
>     "functionName": "getNumberOfRemainingTickets",
>     "contractAddress": "0x494536eF0C3efeCEAd3E119b434393B336230740",
> }
> EOF
{"status":400,"message":"Error: chainId not defined"}
```

Figure 27. Curl request to query smart contract with a missing parameter

Validation result: Pass

Functional requirement 4.4

Acceptance criteria: The response from the oracle should contain the requested data.

Evaluation details: Figure 28 shows the curl request and the reply of the oracle. When this request is made, the lottery has been initialized with a total number of 10 tickets.

The figure shows that the value is 10, which is the expected value when no lottery tickets have been bought yet.

```
$ curl \
> https://DESKTOP-FQK0S45.local:3002/callViewFunction \
> --cacert ./ca_certificate.pem \
> --cert offchain_certificate.pem \
> --key offchain_key.pem \
> -d @- <<EOF
>
> {
>     "functionName": "getNumberOfRemainingTickets",
>     "contractAddress": "0x494536eF0C3efeCEAd3E119b434393B336230740",
>     "chainId": 4
> }
> EOF
{"status":200,"value":"10","functionName":"getNumberOfRemainingTickets"}
```

Figure 28. Successful curl request to query the smart contract

Validation result: Pass

5.3.5 Update Smart Contract

In this section, we evaluate the functional requirements related to the smart contract update functionality of the generated oracle application. The associated use case is described in Table 11 and the functional requirements are described in Table 16.

Functional requirement 5.1

Acceptance criteria: The */transaction* endpoint can be seen in the oracle source code and the mapping between an API request's parameters and the transaction created by the oracle can be seen in the source code of the oracle.

Evaluation details: The code for this function is quite big, so the screenshot will not be provided, but the template for the oracle code generated by the model is available in the repository at `https://gitlab.com/olevasseur/mde-blockchain-oracles`, so the reader can verify that there is a */transaction* endpoint that respects the evaluation criteria. In the evaluation scenario, this endpoint is used in buying 5 tickets from an account A.

After buying 5 tickets with account A, account A has 5 tickets and there are 5 remaining tickets to be bought.

Validation result: Pass

Functional requirement 5.2

Acceptance criteria: The reply to the off-chain query is used to create the transaction and it can be seen in the source code of the oracle.

Evaluation details: It can be seen when generating the oracle code from the demo model in the repository that there is a function that queries the off-chain application at the */getNumberOfTickets* endpoint. If the response contains a number of tickets that is larger than 0, then a transaction is made to buy that number of tickets. In the evaluation scenario, this type of transaction constitutes the push-based inbound oracle and is used by account B to buy the 5 remaining tickets. After account B bought 5 tickets, there are 0 remaining tickets and the exchange rate for the lottery prize has not been set yet.

Validation result: Pass

Functional requirement 5.3

Acceptance criteria: The response to an event notification is used to create the transaction and the mapping between the response content and the transaction created by the oracle can be seen in the source code of the oracle.

Evaluation details: It can be seen when generating the oracle application from the demo model in the repository that for the *RequestForExchangeRate* event, a function is made to notify the off-chain application that the event has been emitted. This function uses transaction parameters from the model, which are overwritten by the reply to the event notification if necessary. In **5.2**, the 5 remaining tickets for the lottery have been bought. When this happens, the smart contract triggers a *RequestForExchangeRate* event, which is picked up by the oracle application. The oracle application forwards the event notification to the off-chain application which replies with a transaction request.

Validation result: Pass

Functional requirement 5.4

Acceptance criteria: A response with HTTP status 400 is returned to the client when the

client makes a request with invalid parameters. The response also contains a message field with a description of the error.

Evaluation details: Figure 29 shows an example of an error message returned when the *privateKey* is missing from the request.

```
curl \
> https://DESKTOP-FQK0S45.local:3002/transaction \
> --cacert ./ca_certificate.pem \
> --cert oracle_certificate.pem \
> --key oracle_key.pem \
> -d @- <<EOF
>
> {
>     "functionName": "resetLottery",
>     "contractAddress": "0x494536eF0C3efeCEAd3E119b434393B336230740",
>     "chainId": 4,
>     "_ticketPrice": 10000000000000000,
>     "_ticketLimitPerCustomer": 10,
>     "_lotteryPrize": 500,
>     "_totalNumberOfTickets": 10
> }
> EOF
{"status":400,"message":"Missing transaction attributes: privateKey"}
```

Figure 29. Curl request to make a transaction with a missing parameter

Validation result: Pass

Functional requirement 5.5

Acceptance criteria: Using the oracle, the smart contract can be queried before and after the transaction and it can be seen that the transaction has modified the state of the smart contract.

Evaluation details: Figure 28 showed the number of remaining lottery tickets when the lottery just started. After buying 5 tickets as described in **5.1**, calling the same function returns a value of 5, which is the expected behavior. In regards to the evaluation scenario, the fact that the *RequestForExchangeRate* event was triggered also shows that

the transactions worked properly.

Validation result: Pass

5.3.6 Listen to event notifications

In this section, we evaluate the functional requirements related to the smart contract event subscription functionality of the generated oracle application. The associated use case is described in Table 12 and the functional requirements are described in Table 16.

Functional requirement 6.1

Acceptance criteria: An event instantiation can be seen in the source code with a callback that takes the event object as a parameter.

Evaluation details: Figure 30 shows the event subscription for the *AnnounceWinner* event as well as the callback taking the event as a parameter. In the evaluation scenario, this happens when the exchange rate has been sent to the smart contract. The smart contract then sends the right amount in ETH to the winner and sends an event announcing who the winner is.

Validation result: Pass

Functional requirement 6.2

Acceptance criteria: An HTTPS POST request containing the data of the event can be seen in the source code of the callback of the event detection.

Evaluation details: Figure 30 shows the HTTPS POST sending the content of the event object to the off-chain application.

Validation result: Pass

5.4 Evaluation of Security Requirements

In this section, we evaluate the security requirements of the solution based on their corresponding evaluation criteria. These functional requirements and acceptance criteria are described in Table 17.

```
web3Contracts['0x494536eF0C3efeCEAd3E119b434393B336230740'].events.
    AnnounceWinner({
  filter: {},
  fromBlock: 'latest',
}, async (error, event) => {
  if (!error) {
    console.log('detected event!');
    const returnValues = event.returnValues;
    const data = JSON.stringify(returnValues);
    const options = getPostRequestOptions('localhost', '/
        winnerDeclared', 3001, data)
    const request = https.request(options);
    request.write(data);
    request.on('error', (error) => {
      let destination = options.hostname + ':' + options.port +
          options.path;
      console.log('Error making a request to ' + destination + ": " +
           error);
    });
    request.end();
```

Figure 30. Oracle application code snippet: AnnounceWinner event subscription

Security requirement 1

Acceptance criteria: The MagicDraw plugin must read the certificate with the X509 certificate utility of the OpenSSL library and verify that a successful exit code is returned.

Evaluation details: Looking at the plugin's source code, it is possible to see that the following OpenSSL command is used to read the certificate and that the status code of this command is compared to 0 to see if the certificate file is valid:

```
$ openssl x509 -in certificate.pem -text
```

Validation result: Pass

Security requirement 2

Acceptance criteria: The MagicDraw plugin must read the private key with the appropriate key utility of the OpenSSL library and must verify that a successful exit code is returned. The supported key algorithms are RSA and elliptic curve algorithms.

Evaluation details: Looking at the source code of the plugin, it is possible to see that the OpenSSL command below is used to read RSA private keys. Alternatively, to read an

elliptical curve private key, the *rsa* part of the command is replaced with *ec*. Then the status code of this command is compared to 0 to see if the private key file is valid.

```
$ openssl rsa -in privkey.pem -text
```

Validation result: Pass

Security requirement 3

Acceptance criteria: The MagicDraw plugin must use the OpenSSL library to read the size of the private key. The size of the key is at least 3072 bits for RSA keys and at least 256 bits for elliptic curve keys.

Evaluation details: Looking at the source code of the plugin, it is possible to see that the OpenSSL command below is used to get the size of the key from the certificate. The output of that command is then used to extract the key size and compare it to the appropriate acceptable size for the given algorithm.

```
$ openssl x509 -in certificate.pem -text | grep Public-Key
```

Validation result: Pass

Security requirement 4

Acceptance criteria: The MagicDraw plugin must use the OpenSSL library to output the public key from the certificate and from the private key and must verify that the two outputs are identical.

Evaluation details: Looking at the source code of the plugin, it is possible to see that the following OpenSSL commands are used to get the public key from the certificate and from the private key respectively. Note that if the private key uses elliptic curve cryptography, then *rsa* is replaced with *ec* in the second command. The output of these two commands is then compared to make sure that the private key matches the certificate.

```
$ openssl x509 -in certificate.pem -pubkey -noout
$ openssl rsa -in privkey.pem -pubout
```

Validation result: Pass

Security requirement 5

Acceptance criteria: A client's request to the oracle API should fail if it does not use HTTPS.

Evaluation details: Figure 31 shows that the generated oracle returns an empty reply when making an unsecure HTTP request.

```
curl \
> http://DESKTOP-FQK0S45.local:3002/callViewFunction \
> --cacert ./ca_certificate.pem \
> --cert offchain_certificate.pem \
> --key offchain_key.pem \
> -d @- <<EOF
>
> {
>     "functionName": "getNumberOfRemainingTickets",
>     "contractAddress": "0x494536eF0C3efeCEAd3E119b434393B336230740",
>     "chainId": 4
> }
> EOF
curl: (52) Empty reply from server
```

Figure 31. Empty reply from oracle for insecure HTTP request

Validation result: Pass

Security requirement 6

Acceptance criteria: A client's request to the oracle API should fail if the certificate provided is not trusted by the oracle.

Evaluation details: Figure 32 shows that the generated oracle returns an error when making a request with an untrusted certificate.

Validation result: Pass

Security requirement 7

Acceptance criteria: The source code of the oracle server contains a middleware component that verifies the key size of the key in the client's certificate. The requirements for

```
curl \
> http://DESKTOP-FQK0S45.local:3002/callViewFunction \
> --cacert ./ca_certificate.pem \
> --cert badcert.pem \
> --key badkey.pem \
> -d @- <<EOF
>
> {
>     "functionName": "getNumberOfRemainingTickets",
>     "contractAddress": "0x494536eF0C3efeCEAd3E119b434393B336230740",
>     "chainId": 4
> }
> EOF
curl: (56) OpenSSL SSL_read: Connection reset by peer, errno 104
```

Figure 32. Error from the server when making request with an untrusted certificate

the client's key size are the same than for the server's key size.

Evaluation details: Figure 33 shows that the generated oracle returns an error when making a request with a key whose size does not respect the requirements.

Validation result: Pass

5.5 Answers to Research Questions

In this section, we answer research question **[RQ4] What is the usability of the proposed solution for MDE of blockchain oracles?** An evaluation scenario has been described. Based on this evaluation scenario, a smart contract has been written, a simple off-chain application has been written for testing purposes and four different oracles have been modeled. From the model of the oracles, an oracle application has been generated using the MagicDraw plugin developed in section 3. This oracle application has been evaluated based on the functional requirements and the security requirements and their associated evaluation criteria.

[RQ4.1] What would be a proper evaluation scenario for the contribution? A proper evaluation scenario is a scenario that offers the possibility to model the four types of oracles described in this thesis. It is also a scenario that allows verifying all of the functional and security requirements. The lottery application suggested in this section meets those requirements. **[RQ4.2] Does the solution satisfy the functional require-**

```
curl \
> http://DESKTOP-FQK0S45.local:3002/callViewFunction \
> --cacert ./ca_certificate.pem \
> --cert offchain_ec_certificate.pem \
> --key offchain_ec_key.pem \
> -d @- <<EOF
>
> {
>     "functionName": "getNumberOfRemainingTickets",
>     "contractAddress": "0x494536eF0C3efeCEAd3E119b434393B336230740",
>     "chainId": 4
> }
> EOF
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Error</title>
</head>
<body>
<pre>Error: Unauthorized ...
```

Figure 33. Error from the server when key size does not respect requirements

ments? All of the functional requirements described in this section along with their acceptance criteria have been met, so the solution satisfies the functional requirements. **[RQ4.3] Does the solution satisfy the security requirements?** All of the security requirements described in this section along with their acceptance criteria have been met, so the solution satisfies the security requirements.

6 Concluding remarks

This thesis aims at answering one main research question: **How does model-driven engineering facilitate the modeling of secure blockchain oracles?** To answer this question, a literature review is first conducted to identify the main elements modeled by existing solutions and identify the strengths and weaknesses of these solutions. The literature review shows that while some elements of blockchain applications are relatively well covered by the existing literature, there has not been much research about the model-driven engineering of blockchain oracles. To make up for this limitation, we suggest a UML profile that extends the UML sequence diagram with elements specific to the blockchain oracle domain. Additionally, a MagicDraw plugin is developed to transform the model of the oracles into an executable and secure Node.js application. Finally, we validate our contribution by identifying functional requirements, security requirements, and their associated evaluation criteria. A lottery application is designed as an evaluation scenario that offers the possibility to model four types of oracles and cover all of the functional requirements and security requirements.

The answer to the main research question is that model-driven engineering facilitates the modeling of secure blockchain applications by providing a model that is adapted and consistent with the blockchain oracle domain. This model then makes it easy to map the model elements to the source code and automate this transformation. The successful generation of a functional blockchain oracle application confirms that the model fits the needs of the oracles. As for the security part, the integration of mandatory security features in the model ensures that the required security features are taken into consideration at the design level and that they will be present in the generated source code.

6.1 Limitations

There are a few limitations to our literature review that we discuss as threats to validity [ZJZ+16]. The restricted time span refers to the researchers' incapacity to foresee relevant studies outside of the time frame planned during the SLR planning phase. In addition, some of the modeling solutions may have been modified or improved between the time of this work. This limitation is a reality of SLRs and cannot really be mitigated. The bias in study selection is another threat to the validity of this literature review. This occurs when researchers have their own subjective conjecture and do not apply inclusion and exclusion criteria consistently or use incompatible search terms. An attempt to counter this limitation is made by gathering feedback from other researchers in the field to include any missing relevant work in the SLR. The bias in data extraction and subjective interpretation are the last threats to validity that we have identified for this literature review. These happen when researchers have different interpretations and opinions about the extracted data. To mitigate these biases, researchers involved in the

SLR to share their points of view and discuss until they reach a consensus.

Regarding the contribution section of this thesis, we have identified 2 main limitations. The oracle model suggested in this thesis is meant to be intuitive and usable by people who are not blockchain experts. However, for security purposes, the MagicDraw plugin makes it mandatory for its users to configure digital certificates. This feature limits the usability of the plugin to users who have a basic understanding of digital certificates, which from our experience is not a very intuitive concept. Another limitation of this contribution is the support and maintenance of the plugin. Since the plugin must read the model before generating the source code, in order to add a simple detail like a new tagged value for a stereotype, the source code of the plugin has to be modified and recompiled every time, which is not very convenient.

Finally, regarding the evaluation section of this thesis, we have identified one main limitation. The requirements and evaluation criteria of the contribution were written by the authors of this thesis. It is possible that they do not exactly reflect what other engineers in the field would expect from a model-driven engineering solution for blockchain oracles.

6.2 Future work

Future work includes support for other types of blockchain platforms. One of the objectives of MDE is to be able to support multiple platforms using a common abstract syntax [WK04]. In this thesis, the Ethereum platform is supported, but the same concepts could possibly be applied to a different blockchain platform (e.g., Hyperledger Fabric). This can help validate the model of the oracles suggested in this thesis and can make the plugin developed in this thesis even more useful. Alternatively, support for other types of encryption schemes can be supported for the communication with the oracle. This can provide a larger choice of encryption schemes which can make the oracle application available to more clients and provide different options in case some encryption schemes become outdated or are found to contain vulnerabilities. Finally, the API of the oracle can possibly be extended to allow users to create event registrations through HTTP requests without having to create them through the model. This can be used to avoid re-generating the source code every time a new oracle needs to be created and can increase the usability of the generated oracle application by making it possible for off-chain applications to create oracles in real-time.

References

[Arc] Archimate 3.1 specification. https://pubs.opengroup.org/architecture/archimate3-doc/toc.html. Accessed: 2022-03-05.

[BK20] Eduard Babkin and Nataliya Komleva. Model-driven liaison of organization modeling approaches and blockchain platforms. In *Advances in Enterprise Engineering XIII*, pages 167–186. Springer International Publishing, 2020.

[BPRBM21] Juan Boubeta-Puig, Jesús Rosa-Bilbao, and Jan Mendling. CEPchain: A graphical model-driven solution for integrating complex event processing and blockchain. *Expert Systems with Applications*, 184:427–435, 12 2021.

[CMM+20] F. Corradini, A. Marcelletti, A. Morichetta, A. Polini, B. Re, and F. Tiezzi. Engineering trustable choreography-based systems using blockchain. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*. ACM, March 2020.

[CMO18] Giovanni Ciatto, Stefano Mariani, and Andrea Omicini. Blockchain for trustworthy coordination: A first study with LINDA and ethereum. In *2018 IEEE/WIC/ACM International Conference on Web Intelligence (WI)*. IEEE, December 2018.

[FHB+19] Ghareeb Falazi, Michael Hahn, Uwe Breitenbucher, Frank Leymann, and Vladimir Yussupov. Process-based composition of permissioned and permissionless blockchain smart contracts. In *2019 IEEE 23rd International Enterprise Distributed Object Computing Conference (EDOC)*. IEEE, October 2019.

[FN16] Christopher K. Frantz and Mariusz Nowostawski. 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, September 2016.

[Fou] Ethereum Foundation. On slow and fast block times. https://blog.ethereum.org/2015/09/14/on-slow-and-fast-block-times/.

[GD19] Luca Guida and Florian Daniel. Supporting reuse of smart contracts through service orientation and assisted development. In *2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)*. IEEE, April 2019.

[GKGK18] Peter Garamvolgyi, Imre Kocsis, Benjamin Gehl, and Attila Klenik. 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, June 2018.

[He20] Xudong He. Modeling and analyzing smart contracts using predicate transition nets. In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, December 2020.

[HMQ20] Mohammad Hamdaqa, Lucas Alberto Pineda Metz, and Ilham Qasse. iContractML. In *Proceedings of the 12th System Analysis and Modelling Conference*. ACM, October 2020.

[HSP18] Barbora Hornáčková, Marek Skotnica, and Robert Pergl. Exploring a role of blockchain smart contracts in enterprise engineering. In *Advances in Enterprise Engineering XII*, pages 113–127. Springer International Publishing, December 2018.

[HZD+20] Kai Hu, Jian Zhu, Yi Ding, Xiaomin Bai, and Jiehua Huang. Smart contract engineering. *Electronics*, 9(12):2042, December 2020.

[IM19] Mubashar Iqbal and Raimundas Matulevičius. Blockchain-based application security risks: A systematic literature review. In *Lecture Notes in Business Information Processing*, pages 176–188. Springer International Publishing, 2019.

[LPGBD+19] Orlenys López-Pintado, Luciano García-Bañuelos, Marlon Dumas, Ingo Weber, and Alexander Ponomarev. Caterpillar: A business process execution engine on the ethereum blockchain. *Software: Practice and Experience*, May 2019.

[LTW+20] Qinghua Lu, An Binh Tran, Ingo Weber, Hugo O'Connor, Paul Rimba, Xiwei Xu, Mark Staples, Liming Zhu, and Ross Jeffery. Integrated model-driven engineering of blockchain applications for business processes and asset management. *Software: Practice and Experience*, 51(5):1059–1079, November 2020.

[LWW19] Jan Ladleif, Mathias Weske, and Ingo Weber. Modeling and enforcing blockchain-based choreographies. In *Lecture Notes in Computer Science*, pages 69–85. Springer International Publishing, 2019.

[LWW20] Jan Ladleif, Ingo Weber, and Mathias Weske. External data monitoring using oracles in blockchain-based process execution. In *Lecture Notes in Business Information Processing*, pages 67–81. Springer International Publishing, 2020.

[LXSY20] Sin Kuang Lo, Xiwei Xu, Mark Staples, and Lina Yao. Reliability analysis for blockchain oracles. *Computers & Electrical Engineering*, 83:106582, May 2020.

[MA19] Yvonne Murray and David A. Anisi. Survey of formal verification methods for smart contracts on blockchain. In *2019 10th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. IEEE, June 2019.

[MBF+20] Roman Mühlberger, Stefan Bachhofner, Eduardo Castelló Ferrer, Claudio Di Ciccio, Ingo Weber, Maximilian Wöhrer, and Uwe Zdun. Foundational oracle patterns: Connecting blockchain to the off-chain world. In *Lecture Notes in Business Information Processing*, pages 35–51. Springer International Publishing, 2020.

[MBH18] Lucie Mercenne, Kei-Leo Brousmiche, and Elyes Ben Hamida. Blockchain studio: A role-based business workflows management system. In *2018 IEEE 9th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*. IEEE, November 2018.

[ML18] Anastasia Mavridou and Aron Laszka. Designing secure ethereum smart contracts: A finite state machine based approach. In *Financial Cryptography and Data Security*, pages 523–540. Springer Berlin Heidelberg, 2018.

[MMT18] Michele Marchesi, Lodovica Marchesi, and Roberto Tonelli. An agile software engineering method to design blockchain applications. In *Proceedings of the 14th Central and Eastern European Software Engineering Conference Russia on ZZZ - CEE-SECR '18*. ACM Press, 2018.

[PXB+19] Hye-Young Paik, Xiwei Xu, H. M. N. Dilum Bandara, Sung Une Lee, and Sin Kuang Lo. Analysis of data management in blockchain-based systems: From architecture to governance. *IEEE Access*, 7:186091–186107, 2019.

[RD18] Henrique Rocha and Stéphane Ducasse. Preliminary steps towards modeling blockchain oriented software. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*. ACM, May 2018.

[SGS18] Diogo Silva, Sérgio Guerreiro, and Pedro Sousa. Decentralized enforcement of business process control using blockchain. In *Advances in Enterprise Engineering XII*, pages 69–87. Springer International Publishing, December 2018.

[TB20] Nguyen Khoi Tran and M. Ali Babar. Anatomy, concept, and design space of blockchain networks. In *2020 IEEE International Conference on Software Architecture (ICSA)*. IEEE, March 2020.

[Wik] Bitcoin Wiki. Confirmation [of bitcoin transactions]. `https://en.bitcoin.it/wiki/Confirmation`.

[WK04] Jos B. Warmer and Anneke G. Kleppe. *The object constraint language: Getting your models ready for MDA*. Addison Wesley Longman, 2004.

[WXR+16] Ingo Weber, Xiwei Xu, Régis Riveret, Guido Governatori, Alexander Ponomarev, and Jan Mendling. Untrusted business process monitoring and execution using blockchain. In *Lecture Notes in Computer Science*, pages 329–347. Springer International Publishing, 2016.

[XLL+19] Xiwei Xu, Qinghua Lu, Yue Liu, Liming Zhu, Haonan Yao, and Athanasios V. Vasilakos. Designing blockchain-based applications a case study for imported product traceability. *Future Generation Computer Systems*, 92:399–406, March 2019.

[XPZ+18] Xiwei Xu, Cesare Pautasso, Liming Zhu, Qinghua Lu, and Ingo Weber. A pattern collection for blockchain-based applications. In *Proceedings of the 23rd European Conference on Pattern Languages of Programs*. ACM, July 2018.

[ZJZ+16] Xin Zhou, Yuqin Jin, He Zhang, Shanshan Li, and Xin Huang. A map of threats to validity of systematic literature reviews in software engineering. In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2016.

Appendix

I. Licence

Non-exclusive license to reproduce thesis and make thesis public

I, **Olivier Levasseur**,
*(*author's name)

1. herewith grant the University of Tartu a free permit (non-exclusive license) to

   reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   Model-Driven Engineering of Blockchain Oracles,
   *(*title of thesis)

   supervised by Mubashar Iqbal and Raimundas Matulevičius.
   *(*supervisor's name)

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons license CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive license does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Olivier Levasseur
17/05/2022