MANTAS JURGELAITIS

MDA-BASED METHOD FOR DEVELOPMENT OF SMART CONTRACT-BASED SYSTEMS

DOCTORAL DISSERTATION

Kaunas
2023

KAUNAS UNIVERSITY OF TECHNOLOGY

MANTAS JURGELAITIS

MDA-BASED METHOD FOR DEVELOPMENT OF SMART CONTRACT-BASED SYSTEMS

Doctoral dissertation
Technological Sciences, Informatics Engineering (T 007)

2023, Kaunas

This doctoral dissertation was prepared at Kaunas University of Technology, Faculty of Informatics, Department of Information Systems during the period of 2018–2022.

The doctoral right has been granted to Kaunas University of Technology together with Vilnius Gediminas Technical University.

Scientific Supervisor:
Assoc. Prof. Dr. Rita BUTKIENĖ (Kaunas University of Technology, Technological Sciences, Informatics Engineering, T 007).

Edited by: English language editor Dr. Armandas Rumšas (Publishing House *Technologija*), Lithuanian language editor Aurelija Gražina Rukšaitė (Publishing House *Technologija*)

Dissertation Defense Board of Informatics Engineering Science Field:
Prof. Dr. Tomas BLAŽAUSKAS (Kaunas University of Technology, Technological Sciences, Informatics Engineering, T 007) – **chairperson**;
Assoc. Prof. Dr. Nikolaj GORANIN (Vilnius Gediminas Technical University, Technological Sciences, Informatics Engineering, T 007);
Prof. Dr. Raimundas MATULEVIČIUS (Tartu University, Estonia, Technological Sciences, Informatics Engineering, T 007);
Prof. Dr. Simona RAMANAUSKAITĖ (Vilnius Gediminas Technical University, Technological Sciences, Informatics Engineering, T 007);
Prof. Dr. Tomas SKERSYS (Kaunas University of Technology, Technological Sciences, Informatics Engineering, T 007).

The official defense of the dissertation will be held at 11:30 a.m. on 27 June, 2023 at the public meeting of Dissertation Defense Board of Informatics Engineering Science Field in M7 Hall at The Campus Library of Kaunas University of Technology.

Address: Studentų 48–M7, Kaunas, LT-51365, Lithuania.
Phone: (+370) 608 28 527; e-mail doktorantura@ktu.lt

The doctoral dissertation was sent out on 26 May, 2023.
The doctoral dissertation is available on the internet at http://ktu.edu, at the libraries of Kaunas University of Technology (Donelaičio 20, Kaunas, LT-44239, Lithuania) and Vilnius Gediminas University of Technology (Saulėtekio 14, Vilnius, LT-10223, Lithuania).

© M. Jurgelaitis, 2023

KAUNO TECHNOLOGIJOS UNIVERSITETAS

MANTAS JURGELAITIS

MDA ARCHITEKTŪRA GRINDŽIAMAS METODAS IŠMANIŲJŲ KONTRAKTŲ TECHNOLOGIJA GRINDŽIAMOMS SISTEMOMS KURTI

Daktaro disertacija
Technologijos mokslai, informatikos inžinerija (T 007)

Kaunas, 2023

Disertacija rengta 2018–2022 metais Kauno technologijos universiteto Informatikos fakultete, Informacijos sistemų katedroje.

Doktorantūros teisė Kauno technologijos universitetui suteikta kartu su Vilniaus Gedimino technikos universitetu.

Mokslinis vadovas:
doc. dr. Rita BUTKIENĖ (Kauno technologijos universitetas, technologijos mokslai, informatikos inžinerija, T 007).

Redagavo: anglų kalbos redaktorius dr. Armandas Rumšas (leidykla „Technologija“), lietuvių kalbos redaktorė Aurelija Gražina Rukšaitė (leidykla „Technologija“)

Informatikos inžinerijos mokslo krypties disertacijos gynimo taryba:
prof. dr. Tomas BLAŽAUSKAS (Kauno technologijos universitetas, technologijos mokslai, informatikos inžinerija, T 007) – **pirmininkas**;
doc. dr. Nikolaj GORANIN (Vilniaus Gedimino technikos universitetas, technologijos mokslai, informatikos inžinerija, T 007);
prof. dr. Raimundas MATULEVIČIUS (Tartu universitetas, Estija, technologijos mokslai, informatikos inžinerija, T 007);
prof. dr. Simona RAMANAUSKAITĖ (Vilniaus Gedimino technikos universitetas, technologijos mokslai, informatikos inžinerija, T 007);
prof. dr. Tomas SKERSYS (Kauno technologijos universitetas, technologijos mokslai, informatikos inžinerija, T 007).

Disertacija bus ginama viešame Informatikos inžinerijos mokslo krypties disertacijos gynimo tarybos posėdyje 2023 m. birželio 27 d. 11:30 val. Kauno technologijos universiteto Studentų miestelio bibliotekoje, salėje M7.

Adresas: Studentų g. 48-M7, Kaunas, LT-51365, Lietuva.
Tel. (+370) 608 28 527; el. paštas doktorantura@ktu.lt

Disertacija išsiųsta 2023 m. gegužės 26 d.
Su disertacija galima susipažinti interneto svetainėje http://ktu.edu, Kauno technologijos universiteto (K. Donelaičio g. 20, Kaunas, LT-44239, Lietuva) ir Vilniaus Gedimino technikos universiteto (Saulėtekio al. 14, Vilnius, LT-10223, Lietuva) bibliotekose.

© M. Jurgelaitis, 2023

TABLE OF CONTENTS

TABLE OF CONTENTS 5
LIST OF THE TABLES 8
LIST OF FIGURES 9
LIST OF ABBREVIATIONS AND TERMS 11
INTRODUCTION 12
Motivation 12
Research object and scope 12
Problem Statement and research questions 13
The aim and objectives 13
Research Methodology 13
Defended statements 14
Scientific novelty 15
Practical significance 15
Scientific approbation 15
Structure of the dissertation 15
I. ANALYSIS OF CURRENTLY AVAILABLE METHODS AND SOLUTIONS 17
1.1. Blockchain 17
1.2. Smart Contracts 22
1.2.1. Application of Blockchain Technologies 22
1.2.2. Blockchain technology platforms 23
1.2.3. Smart Contract Development Standards 25
1.2.4. Blockchain technology-based and smart contract-based system 25
1.3. Smart contract-based System development 27
1.4. Modelling in the software development process 30
1.4.1. Modelling Languages 31
1.4.2. Modelling Language Extension Mechanisms 34
1.4.3. Model Validation 36
1.4.4. Model Transformation 36
1.5. Modelling in Smart contract-based System Development 38
1.5.1. Model-Based Engineering approaches 38
1.5.2. Model-Driven Development approaches 40
1.5.3. Model-Driven Architecture approaches and their application in Blockchain-based system development 46
1.6. Analysis of related works by Lithuanian researchers 56
1.7. Analysis summary 57
II. THE PROPOSED MDA-BASED METHOD FOR DEVELOPMENT OF SMART CONTRACT-BASED SYSTEMS 59
2.1. Blockchain CIM 60
2.1.1. Blockchain CIM UML profile 61
2.1.2. Blockchain CIM Definition 62
2.2. Blockchain PIM 64

2.2.1. Blockchain PIM UML profile ... 64
2.2.2. Blockchain PIM Definition ... 66
2.3. Blockchain PSM ... 69
2.3.1. Blockchain PSM UML profile ... 70
2.3.2. Blockchain PSM Definition ... 74
2.4. Transformation of Blockchain PSM to Smart Contract Code ... 83
2.4.1. Smart Contract Metamodel ... 83
2.4.2. Smart Contract Development ... 85
2.5. Method extension ... 92
2.6. Limitations of the method ... 94
III. EXPERIMENTAL EVALUATION ... 95
3.1. Generated smart contract code evaluation ... 95
3.1.1. Ethereum Solidity Smart Contracts ... 95
3.1.2. Hyperledger Fabric Go Chaincodes ... 103
3.2. MDA-based method for smart contract-based system development evaluation ... 109
3.2.1. Blockchain CIM Definition ... 109
3.2.2. Blockchain CIM to Blockchain PIM transformation ... 111
3.2.3. Blockchain PIM to Blockchain PSM Transformation Results ... 113
3.2.4. Blockchain PSM to code transformation results ... 117
3.2.5. Execution of smart contract code ... 118
3.3. MDA-based method for development of smart contract-based systems comparison ... 120
3.4. Threats to the validity of the experiment ... 121
IV. CONCLUSIONS ... 123
SUMMARY ... 125
ĮVADAS ... 125
Motyvacija ... 125
Tyrimo sritis ir objektas ... 125
Sprendžiama problema ir tyrimo klausimai ... 126
Tyrimo tikslas ir uždaviniai ... 126
Tyrimų metodika ... 126
Ginamieji teiginiai ... 127
Mokslinis naujumas ... 128
Praktinė reikšmė ... 128
Rezultatų aprobavimas ... 128
Disertacijos dokumento struktūra ... 128
1. EGZISTUOJANČIŲ SPRENDIMŲ IR METODŲ ANALIZĖ ... 129
Blokų grandinės apibrėžimas ... 129
Išmanusis kontraktas ... 129
Išmaniųjų kontraktų technologija grindžiama sistema ... 130
Išmaniųjų kontraktų technologija grindžiamų sistemų kūrimas ... 130
Modeliais grindžiamų sistemų kūrimo metodų taikymas kuriant išmaniųjų kontraktų technologija grindžiamas sistemas ... 131
MDA taikymas išmaniųjų kontraktų technologija grindžiamoms sistemoms kurti 131

2. SIŪLOMAS MDA ARCHITEKTŪRA GRINDŽIAMAS METODAS IŠMANIŲJŲ KONTRAKTŲ TECHNOLOGIJA GRINDŽIAMOMS SISTEMOMS KURTI 133
3. EKSPERIMENTINIS TYRIMAS 135
Sugeneruotų išmaniųjų kontraktų kodo įvertinimas 135
Siūlomo MDA grindžiamo metodo taikymas 136
MDA grindžiamų metodų palyginimas 136
4. IŠVADOS 136
REFERENCES 139
SCIENTIFIC PUBLICATION AND CONFERENCE LIST 147
Scientific publications in Periodicals 147
Scientific publications in Conference Proceedings 147
APPENDIXES 148
Appendix 1. Blockchain CIM to PIM Model transformation rules 148
Appendix 2. Blockchain PIM to Ethereum PSM transformation rules 151
Appendix 3. Blockchain PIM to Hyperledger Fabric PSM transformation rules 159
Appendix 4. Ethereum PSM to Solidity smart contract transformation templates 165
Appendix 5. Hyperledger Fabric PSM to Go chaincode transformation templates 168

LIST OF THE TABLES

Table 1. Comparison of Blockchain Technology Platforms ... 23
Table 2. MBE approaches for smart contract-based system development. Structural view ... 43
Table 3. MBE approaches for smart contract-based system development. Behavioural view ... 44
Table 4. CIM utilisation in MDA adjacent approaches for smart contract-based system development ... 48
Table 5. CIM utilisation in MDA-based methods ... 51
Table 6. PIM utilisation in MDA adjacent approaches for smart contract-based system development ... 54
Table 7. PSM utilisation in MDA adjacent approaches for smart contract-based system development ... 56
Table 8. Blockchain CIM stereotypes ... 61
Table 9. Implementation of Blockchain CIM validation rules ... 64
Table 10. Blockchain PIM stereotypes ... 65
Table 11. Blockchain PIM validation rule implementation ... 69
Table 12. Ethereum PSM stereotypes ... 71
Table 13. Hyperledger Fabric PSM stereotypes ... 73
Table 14. Implementation of Ethereum PSM Validation Rules ... 81
Table 15. Implementation of Hyperledger Fabric PSM Validation Rules ... 83
Table 16. Comparison of SimpleAuction Smart Contract Code ... 97
Table 17. Comparison of Purchase Smart Contract code ... 99
Table 18. Comparison of StateMachine Smart Contract ... 102
Table 19. Comparison of SmartContract chaincode code ... 108
Table 20. Blockchain CIM to Blockchain PIM transformation results ... 112
Table 21. Blockchain PIM to Ethereum PSM Transformation Results ... 114
Table 22. Blockchain PIM to Hyperledger Fabric PSM Transformation Results . 116
Table 23. HackChain Solidity smart contract code metrics ... 118
Table 24. HackChain Solidity smart contract gas costs ... 119
Table 25. Execution of the HackChain Go chaincode (ms) ... 119
Table 26. MDA-based approaches comparison for smart contract-based system development ... 120

LIST OF FIGURES

Figure 1. The structure of blockchain based on [8] 18
Figure 2. Smart contract-based System composition............ 26
Figure 3. Software development processes: a)Waterfall, b)Continuous deployment, and c) Smart Contract............ 28
Figure 4. Relationship among model-based approaches [45]............ 31
Figure 5. UML 2.5 diagram types [48]............ 33
Figure 6. MOF Model Hierarchy............ 35
Figure 7. MDA abstraction levels and transitions [92]............ 46
Figure 8. Development of CIM [102] [108] 49
Figure 9. MDA-based method for Development of Smart contract-based Systems 59
Figure 10. Blockchain CIM. Extended UML metamodel 61
Figure 11. Blockchain CIM Profile 62
Figure 12. Detailed Blockchain CIM definition step in the context of the method. 62
Figure 13. List of Blockchain CIM validation rules............ 63
Figure 14. Blockchain PIM. Extended UML metamodel............ 65
Figure 15. Blockchain PIM Profile............ 66
Figure 16. Detailed Blockchain PIM definition step in the context of the method . 66
Figure 17. Blockchain CIM to Blockchain PIM Transformation............ 67
Figure 18. List of Blockchain PIM validation rules 68
Figure 19. Ethereum PSM. Extended UML metamodel............ 70
Figure 20. Ethereum PSM Profile 72
Figure 21. Hyperledger Fabric PSM. Extended UML metamodel............ 73
Figure 22. Hyperledger Fabric PSM Profile............ 74
Figure 23. Detailed Blockchain PSM definition step in the context of the method 75
Figure 24. Blockchain PIM to Ethereum PSM Transformation 76
Figure 25. Blockchain PIM to Hyperledger Fabric PSM Transformation............ 78
Figure 26. List of Ethereum PSM Validation Rules............ 80
Figure 27. List of Hyperledger Fabric PSM validation rules 82
Figure 28. Solidity smart contract metamodel............ 84
Figure 29. Go chaincode metamodel............ 85
Figure 30. Detailed Blockchain PIM definition step in the context of the method . 85
Figure 31. Transformation of Ethereum PSM to Solidity smart contract code 87
Figure 32. Transformation of the InteractionFragment set into the Solidity Function code 88
Figure 33. Transformation of Transition Operation into the Solidity Function code 88
Figure 34. Transformation of Transition Sources into the Solidity Function code . 89
Figure 35. Transformation of Transition targets set into the Solidity Function code 89
Figure 36. Transformation of Hyperledger Fabric PSM to Go chaincode 90
Figure 37. Transformation of InteractionFragment set into the Go function code.. 91
Figure 38. Transformation of state machine behaviour into the Go function code . 91
Figure 39. Transformation of Transition sources set into the Go Function code 91

Figure 40. Transformation of Transition target set into the Go Function code 92
Figure 41. MDA-based method implementation 93
Figure 42. Blockchain PIM SimpleAuction smart contract..................................... 96
Figure 43. Ethereum PSM SimpleAuction smart contract 97
Figure 44. Blockchain PIM Purchase smart contract 98
Figure 45. Ethereum PSM Purchase smart contract 99
Figure 46. Blockchain PIM StateMachine smart contract 101
Figure 47. Ethereum PSM StateMachine smart contract..................................... 101
Figure 48. Blockchain PIM SmartContract contract: Class diagram..................... 104
Figure 49. Hyperledger Fabric PSM SmartContract chaincode: Class diagram.... 104
Figure 50. Hyperledger Fabric PSM SmartContract chaincode AssetExists function: Sequence diagram.. 105
Figure 51. Hyperledger Fabric PSM SmartContract chaincode CreateAsset function: Sequence diagram.. 105
Figure 52. Hyperledger Fabric PSM SmartContract chaincode ReadAsset function: Sequence diagram.. 106
Figure 53. Hyperledger Fabric PSM SmartContract chaincode UpdateAsset function: Sequence diagram 106
Figure 54. Hyperledger Fabric PSM SmartContract chaincode DeleteAsset function: Sequence diagram.. 107
Figure 55. Hyperledger Fabric PSM SmartContract chaincode TransferAsset function: Sequence diagram 107
Figure 56. Hyperledger Fabric PSM SmartContract chaincode GetAllAssets function: Sequence diagram 108
Figure 57. Blockchain CIM Issue certificate business process: Activity diagram 110
Figure 58. Blockchain CIM Evaluate submitted solutions business process: Activity diagram 110
Figure 59. Blockchain CIM HackChain use case model: Use case diagram......... 110
Figure 60. Blockchain CIM HackChain domain model: class diagram 111
Figure 61. Blockchain CIM to Blockchain PIM Transformation result: Class diagram 111
Figure 62. Blockchain PIM HackChain smart contract: Class diagram 113
Figure 63. Blockchain PIM HackChain smart contract: State Machine Diagram . 113
Figure 64. Ethereum PSM HackChain smart contract: Class diagram.................. 115
Figure 65. Ethereum PSM HackChain smart contract: State Machine Diagram... 115
Figure 66. Hyperledger Fabric PSM HackChain smart contract: Class diagram .. 117
Figure 67. Hyperledger Fabric PSM HackChain smart contract: State Machine Diagram 117

LIST OF ABBREVIATIONS AND TERMS

Abbreviations:
ABCDE – Agile Block Chain DApp Engineering
AST – Abstract Syntax Tree
ATL – ATLAS Transformation Language
BPMN – Business Process Model and Notation
CIM – Computation Independent Model
CPS – Cyber-Physical Systems
DApp – Decentralized application
DLTs – Distributed Ledger Technologies
DSL – Domain-Specific Languages
ER – Entity Relationship
EVM – Ethereum Virtual Machine
FSM – Finite-State Machines
MBE – Model-Based Engineering
MDA – Model-Driven Architecture
MDD – Model-Driven Development
MDE – Model-Driven Engineering
MOF – Meta-Object Facility
MTL – Model-to-Text Transformation Language
OMG – Object Management Group
PIM – Platform Independent Model
PoS – Proof of stake
PoW – Proof of work
PSM – Platform-Specific Model
SDLC – Software Development Lifecycle
SysML – Systems Modeling Language
SWRL – Semantic Web Rule Language
UML – Unified Modeling Language
XMI – XML Metadata Interchange

INTRODUCTION

Motivation

Blockchain provides a decentralised, distributed database that, together with smart contracts, can be used to enable the decentralisation of business processes. The application of blockchain technologies could be utilised to build a wide variety of solutions to promote trust, enforce auditability, transparency, and reduce reliance on centralised authorities. Unfortunately, the adoption of the blockchain technology is relatively slow for the development of such custom solutions, as the established technologies are still evolving.

The current development of smart contract-based systems mainly deals with the development of smart contracts. The main issue is that, due to the immutability of the blockchain, the developed smart contract – once deployed – cannot be changed, requires a shift of focus to the earlier stages of development in the context of the smart contract and the smart contract-based system development. Coupled with the fact that no established formalised development method exists, this makes the development quite ambiguous and difficult to manage, which results in the prevalence of an unstructured approach. Most development phases require comprehensive analysis, which is, more often than not, unsupported by guidelines and standards. As a result, in order to determine the system requirements and analyse the suitability of the blockchain platform for specific needs, developers need to rely on experience. The implementation phase receives the most support in the form of specific platform community-driven implementation standards, although this makes code cloning prevalent in the development process.

In order to promote blockchain adoption, more general support for other phases of development needs to be provided for those interested in the domain. The development of more custom smart contract-based solutions could be supported by the adoption of a specialised modelling approach for facilitating the understanding of the blockchain technology, by providing the capability to describe systems in a more general language, even producing implementation artefacts, and enabling the support of multiple platforms.

Several model-driven development approaches have been proposed to support blockchain and smart contract development. These methods mainly support code generation and validation activities, but they do not cover the entire life cycle of software development. The most common code generation, for which the methods are tailored, is the *Solidity* programming language for the *Ethereum* platform, followed by the *Hyperledger Fabric* platform. For modelling, the behaviour state machine or adjacent notations are mostly used, additionally, the smart contract structure is outlined by using a class diagram.

Research object and scope

The dissertation research object is the smart contract-based system and smart contract development process, tools, methods, and technologies. The scope includes

two research areas, namely, the smart contract-based system and smart contract development methods and tools, and the application of model-driven software development methods and tools.

Problem Statement and research questions

The current development of smart contract-based systems is complicated. Insufficient support for requirement specification, design, and smart contract code production for the developers is the driving issue for this research. The aforementioned development activities could be supported by a modelling approach which would enable the automated production of the implementation code. The automation of the smart contract-based system development could facilitate the development; therefore, this dissertation intends to answer these research questions:

- Can smart contract-based system development be automated?
- Can the modelling approach support automation of the smart contract-based system development process, and, if so, how?
- Can conceptual smart contract-based system models be transformed into implementation artefacts, and, if so, how?
- What composition of conceptual smart contract-based system models is required to support the production of executable implementation artefacts?

The aim and objectives

The aim of this dissertation is to extend the development capabilities of smart contract-based systems by providing means for automated code generation from conceptual smart contract-based system models. The following objectives are outlined:

1. To analyse blockchain and smart contract technologies, smart contract-based systems and smart contract development processes, model-driven software development;
2. To determine the links between model-driven software system development and smart contract-based systems development;
3. To propose a method for model-driven software system development for smart contract-based systems;
4. To define rules for the transformation of blockchain and smart contract conceptual models into implementation artefacts;
5. To implement the proposed method for automating model-driven software system development of smart contract-based systems;
6. To assess the application of the model-driven software development method in the blockchain development process.

Research Methodology

The research was carried out by using the *constructive research* method [1]. Following the outlined method, the research was carried out in the following steps.

1. **Research problem definition.** Exploratory research analysis on blockchain and smart contract technologies, smart contract-based system development, and a

research problem concerning smart contract-based system development has been outlined.
2. **Potential of research.** The area of modelling-based system development processes, tools, and approaches application has been analysed to determine the potential avenues of improvement.
3. **Problem domain analysis.** Comparative analysis of the modelling application for the development of smart contract-based systems and smart contracts has been performed.
4. **Solution conception.** A proposal for a Model Driven Architecture (MDA) based method for the development of smart contract-based systems has been outlined, encompassing the Computation Independent Model (CIM), the Platform Independent Model (PIM), the Platform Specific Model (PSM) abstraction layers, model transformations, and the overall model development processes.
5. **Implementation and evaluation.** The implementation of the MDA-based method for the development of the smart contract-based system has been experimentally studied, thereby evaluating the produced smart contract artefacts by using model transformations of the implemented method.
6. **Evaluation of the application of the solution.** The feasibility of the application in the overall process of smart contract-based system development has been carried out by employing the method to model a solution for issuing a hackathon certificate issuing solution.
7. **Theoretical relevance analysis.** The outlined proposal has been evaluated in the context of the other MDA-based methods for smart contract-based systems and smart contract development.

Defended statements

1. Smart contract-based system development can be automated by adapting the MDA to support the development process in terms of requirement elicitation, design, and smart contract code production activities.
2. The proposed UML notation extensions can be used to model smart contract-based systems: 1) to specify business processes by using an activity diagram, system requirements using the use case, class diagrams at the CIM abstraction level, 2) to outline the smart contract structural and behavioural features at the PIM abstraction level by using class and state machine diagrams, 3) and additionally, to specify function behaviour by using sequence diagrams, opaque behaviour specifications at a PSM abstraction level. The developed UML extensions also enable model transformations utilised to support the transition between different abstraction-level models and used to extend the smart contract structure by using behavioural details specified in a higher abstraction layer model.
3. The PSM abstraction level smart contract structure and behaviour specified by using UML Class, State Machine, and Sequence diagrams can be used to automate the executable smart contract code generation for *Ethereum* and *Hyperledger Fabric* platforms by using model-to-text transformations.

Scientific novelty

The scientific novelty of this research can be summarised as follows:

1. The proposed MDA-based method utilises UML and outlines UML extensions for the development of smart contract-based systems, which allows specifying smart contracts encompassing not only structural but also behavioural aspects.
2. The proposed MDA-based method employs the MDA framework for the development of smart contract-based systems and utilises three different abstraction level models, encompassing UML Activity, Class, Use Case, State Machine, and Sequence diagrams, to support modelling during the requirements, design, and implementation development phases.
3. The proposed MDA-based method for the transition between different abstraction levels, namely, the Blockchain CIM, Blockchain PIM, and Blockchain PSM, employs model-to-model transformations. Additionally, model-to-text transformations are employed for the production of executable Solidity and Go smart contracts.

Practical significance

The practical significance of this research, while closely related to the scientific novelty, can be summarised as follows:

1. The MDA-based method provides a generalized approach for facilitating and automating the development of smart contract-based systems.
2. The MDA-based method supports the requirement and design phases, ultimately resulting in a generated smart contract code that can be used during implementation. The outlined model can also be utilised for the communication and documentation purposes.
3. The MDA-based method supports multiple platforms and can be tailored for different blockchain technology platforms by providing additional Blockchain PSM profile implementations and PIM to PSM, and PSM to code transformation rules.

Scientific approbation

The results of the dissertation have been presented in 4 publications: two publications in periodical scientific journals (IEEE Access Q2 and MDPI Applied Sciences Q2) and three in conference proceedings. The scientific publication list is presented in the SCIENTIFIC PUBLICATION AND CONFERENCE LIST chapter.

Structure of the dissertation

In the first chapter, exploratory research analysis of the blockchain technology, the smart contracts development process and the role of modelling in the software development process is carried out. Additionally, comparative academic literature analysis of model-based approaches for the application in smart contract-based systems and smart contract development processes is presented as well. In the second chapter, a proposal is outlined based on MDA for the development of smart contract-based systems. The chapter further details the contents of each abstraction

level, namely, Blockchain CIM, PIM, and PSM. The experimental evaluation of the proposed MDA method is presented in the third chapter, whereas the dissertation conclusions are outlined in the fourth chapter. Additionally, after the main body of the thesis, the dissertation summary is provided in the Lithuanian language, which is followed by the references, as well as lists of scientific publications and attended conferences. Appendixes are provided at the end of the thesis.

I. ANALYSIS OF CURRENTLY AVAILABLE METHODS AND SOLUTIONS

The analysis of the relevant related scientific literature is presented in this chapter. An overview of the blockchain technology's key concepts, smart contracts, and development platforms is presented. Afterwards, the currently employed smart contract-based system development processes and the key challenges are analysed. Furthermore, the model-based engineering practices, tools, and approaches are explored, and a comparative analysis of the utilisation of model-based engineering practices for the smart contract-based system development is provided.

1.1. Blockchain

The blockchain is "a distributed database that maintains a continuously growing list of data records secured from tampering and revision" [2] originally introduced by Satoshi Nakamoto in 2008 as a form of cryptocurrency called *Bitcoin*. The blockchain is made up of blocks, and each block is made up of transactions. Each block contains a list of transactions, a timestamp and a hash of the previous block in the blockchain. The entire chain of data blocks and transactions contained in a block makes up a public ledger [3].

Basically, the blockchain is a decentralized distributed database that is shared between all the participants in the peer-to-peer network [4]. Blockchain differs from a traditional database in terms of the way how the transaction data is stored. For any data recording to the distributed ledger, verification of transaction data must be performed, and consensus between the participants must be achieved. Once the transaction data have been verified and saved, the data could only be changed by performing another transaction, thus making the data immutable [5]. Blockchain promotes trust among network participants, as it allows data to store data in a decentralized, distributed manner without relying on a centralized node to ensure that the data are correct [6].

Any data record on the blockchain can only be stored by performing a transaction. Any transaction recorded on a blockchain has an ID, a sender, and a receiving party, a timestamp, a transaction amount, and the total balance. Blockchain authentication is based on public key cryptography and transactions between two parties, and each transaction record has both the source and the destination addresses [7]. The key to the blockchain as storage lies in its distribution: the full copy of the ledger is stored on any participating node, instead of a single centralized server. The ledger copy contains the full blockchain, in which the full list of transactions is recorded.

Figure 1 illustrates a blockchain structure, any block header contains a hash to the previous block, or a parent block, thus linking blocks in a chain [8]. The block body is made up of a transaction counter and transactions. The maximum number of transactions a block can contain depends on the size of the block and the size of each transaction.

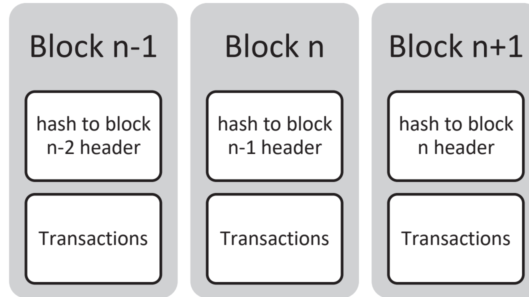


Figure 1. The structure of blockchain based on [8]

The consistency of blockchain data is achieved by verifying the validity of each transaction. Once the transaction has been verified, the transactions are recorded in a block. The blockchain uses an asymmetric cryptography mechanism to validate the authentication of transactions [9]. Each blockchain participant owns a pair of keys: a private key and a public key. When a transaction has been formed, a private key is used to sign it. A digitally signed transaction is then broadcasted to the other peer-to-peer network participants. Thus, each transaction data record insertion is achieved in two phases:

- A signing phase during which a transaction is encrypted by using the private key, and then the transaction is broadcasted.
- A verification phase, during which, the verification of the transaction is performed by using the public key, and thus making sure that the data have not been tampered with.

Ultimately, once a transaction has been recorded in the public blockchain ledger, it is nearly impossible to change the transaction data or add new data before it, as any consecutive blocks would need to be verified once again [7].

The blockchain technology aims to solve several problems of data consistency, transparency, integrity, availability, and verifiability [4] [10] [9] [11] [7]. In a centralized system, users only have limited access to the data, and, considering that the data is not public, it may be subject to tampering by other entities, thus sometimes making the data untrustworthy [12]. Blockchain aims to rectify this by making the data public and shared across all participants, and any data state transitions are public and performed by using consensus algorithms to make sure that the data insertion was executed correctly. Therefore, one of the key differences in blockchain from a traditional database is that, instead of relying on a central authority that provides these validation and storage services, on the blockchain, the services are performed by using consensus [12].

In a blockchain, the public ledger is considered the single source of truth. This is also ensured by the fact that any data modification, in contrast to the 'traditional' storage, cannot be altered after insertion [10]. Any data modification can be achieved only by appending new blocks to the blockchain. The ledger is publicly available to any participants in the peer-to-peer network. Since the data are public, anyone can verify the integrity of the data, which is not always supported by a

centralized system [10]. In essence, the blockchain decentralizes data storage, and, by validating new block insertions to the blockchain while using consensus algorithms, a network is more secure as it makes it harder to forge or modify data.

In a blockchain, all network participants have a copy of the public ledger. This kind of replication enables the data to be validated/verified even if a network node fails [13]. The distribution of data ensures that the public ledger is always available, as long as a participant exists in a network [10]. This makes the blockchain transaction execution slower in principle than a query to a traditional database, as, in each case, a transaction needs to be formed, validated, and then bundled in a block and committed to a blockchain. Still, the technology provides a number of advantages by a few main characteristics with which any blockchain can be identified [4] [7]:

- Distribution. Identical ledger copies that encompass all records are shared by all participants. In a centralized system, each transaction needs to be validated by the central authority, which sometimes results in cost and performance bottlenecks. In the blockchain system, it is the participants who can verify the data, and, since verification is not dependent on a single centralized entity, this comes with a time cost since a consensus needs to be reached in order to save data in a blockchain. Moreover, if any participating node fails, the remaining nodes can continue to operate, thus ensuring higher data availability.
- Public ledger. The history of all transactions is always available in the public ledger. Any data that have been committed to the blockchain are timestamped and can be directly traced to a previous transaction or a specific address. While the verification of a transaction takes longer than in a traditional database, by utilising consensus algorithms, valid transactions are committed to the blockchain, and invalid ones are rejected.
- Immutability. Any transaction data that are recorded to the blockchain cannot be deleted, rollbacked, or changed easily. This is based on the fact that any transaction that is recorded needs to be validated by the network, and, once it has been committed to the block, the miners verify the full block and add it to the chain. Once this has been done, any further modifications to a transaction that exists in a blockchain would need to also be validated. Since validation is costly, there is no incentive to validate the previous blocks. The only way to circumvent this is by forking the blockchain, which creates a side chain which exists independently of the main chain. Unfortunately, if there are no participants, the fork would eventually die [10].
- Anonymity. Blockchain network participants can only interact with the blockchain by using an address. This address is public for all participants, but it does not reveal the participant's identity. This ensures anonymity, which is highly prominent in permissionless or public blockchains.
- Consensus algorithm. Network participants collectively authenticate and approve transactions that are to be appended to the blockchain. There are a number of consensus algorithms, but, in essence, the majority of participants need to agree to the transaction content correctness before it is committed to the blockchain. This is the main mechanism in the blockchain which ensures that

the data is maintained and consistent. Amongst the most popular consensus mechanisms is *proof of work*, and *proof of stake*.

- Proof-of-Work (PoW) ensures the immutability of the transaction records and is the backbone algorithm used by a number of blockchains. During the consensus, miners in the network perform the task of validating the transactions proposed for addition to the blockchain. Miners need to calculate a solution to a one-way hash function. This computation is done mostly by guessing a value that would hash out to an acceptable hash. Unfortunately, these computations do not have any practical usage apart from ensuring that block building is costly, which makes PoW highly inefficient and wastes a lot of energy/electricity in the process [11]; this strategy is used in Bitcoin, Ethereum, Bitcoin Cash, Dogecoin, Litecoin, and other platforms.
- Proof-of-Stake (PoS) for block commitment to the blockchain utilises the ownership of digital assets (stake). The transaction validator who has a higher stake is selected to verify the transactions, and this works under the assumption that participants with a larger share of wealth are more trustworthy [11]. This algorithm is less computationally intensive and is used in Ethereum 2.0, Cardano, and Binance Chain platforms [14].

Another distinction that can be drawn is the particularity of the blockchain when it comes to the availability and access of the network participants, based on which, the blockchain can be devised into three major types [8] [15]:

- Public (Permissionless);
- Private (Permissioned);
- Consortium.

Public or non-permission blockchains [15] can be distinguished by the ability to join and leave the network as a participant at any time. In a permissionless blockchain, no central authority exists which manages roles or permissions; also, every participant is equal, and can read or write data since the data are publicly available. Additionally, in a permissionless blockchain, anonymity is highly utilised, as there is no way to associate blockchain addresses with specific identities. Bitcoin and Ethereum [2] are the most popular public blockchains. Like most public blockchains, they employ proof of work as a consensus algorithm. The high number of network participants allows one to guard against attacks, thus making the network more resilient.

Permissioned or private blockchains [10] [16] authorize a limited number of blockchain readers and writers. In a private blockchain, a central node (organization) distributes and issues the rights for specific participants for the capabilities to commit or read data to and from the blockchain. Privacy is not always ensured, as the participant's identity needs to be known for the management of permissions. The most widely known instance of permissioned or private blockchains is *Hyperledger Fabric*.

A consortium blockchain is considered to be semi-decentralized. Like a private blockchain, it also is permissioned, although, while in a private blockchain, the permissions management relies on a single organization, in a consortium blockchain, this task could be delegated to a number of nodes. The reading rights can be restricted by the management authorities, and a limited number of trusted nodes is needed to reach a consensus [11].

Similarly, as a distinction can be outlined based on the availability of the blockchain ledger and participant permissions management, a division can be drawn between blockchain technologies in the generation [4] [7]. The main difference between different generations is the support to develop custom solutions by using smart contracts. The blockchain – when first introduced – only dealt with cryptocurrency and its exchange capabilities. This made the application limited to other sectors. Meanwhile, the cryptocurrency application has not died down, and newer blockchain technologies also place a heavier focus on blockchain applications in other areas rather than only financial.

Blockchain 1.0 is mainly defined by cryptocurrencies [4]. The Blockchain technology began with *Bitcoin* [2], and many developers still consider it the main blockchain technology. Bitcoin is effective as a ledger for a cryptocurrency, but this simplicity and inability to support more complex contracts or agreements became the main motivation for the emergence of *Blockchain 2.0*. Bitcoin introduced a basic Turing-incomplete scripting language that allowed some form of contractual functionality, but the incomplete support became the main limitation that newer blockchains – such as *Ethereum* – are addressing.

Blockchain 2.0 refers to applications hosted on the blockchain which utilise smart contracts [4]. These applications of smart contracts do not necessarily deal with monetary operations. Blockchains, such as Ethereum, enable developers to deploy their own solutions. These capabilities are provided through smart contracts and are achieved by the introduction of a virtual machine that runs on top of the blockchain. The expanded applicability of the technology allows newer generations of blockchain to support smart contracts to be applied to a number of domains, which was not possible when using the bitcoin. While smart contracts increase the applicability of the blockchain technology, there are a number of areas, such as scalability, interoperability, adaptability, sustainability, privacy, and faster transaction time [17], which need attention for the blockchain to become a more prominent technology.

Furthermore, *Blockchain 3.0* [4] and later emerging solutions utilise the blockchain as a network for communication purposes between autonomous machines for decision-making capabilities. Currently, no defining solutions exist that could describe this sort of blockchain technology, as the majority are still at the proposal stages.

In the scope of this work, blockchains supporting smart contracts are analysed further because they provide capabilities for building custom solutions for decentralizing processes. Additionally, any benefits that the blockchain provides, be it immutability, public ledger, or distributed architecture, have some sort of effect on the development of smart contract-based solutions. Furthermore, since a clear

distinction can be drawn between public and private blockchains in the scope of this work, both should be analysed further, and any sort of proposal must include both ends of the availability spectrum.

1.2. Smart Contracts

As mentioned above, in Blockchain 2.0, the next step in implementing the blockchain technology lies in the support of more complex custom solutions rather than simply cryptocurrencies [18]. This can be achieved by using agreements that are stored, verified, and executed on a blockchain, which are called *smart contracts* [8]. The main purpose of such a contract is to automatically execute the terms of an agreement once certain conditions have been met. Simply stated, it is a computer program that follows an *if this happens, then this* structure [13]. Since smart contracts are built on top of the blockchain, they inherit all of those characteristics of the blockchain, which makes them a great tool for decentralizing various business processes. Although smart contracts could theoretically cover entire software applications, the majority of applications lie in the financial or notary category [19]. Since smart contracts can express triggers, and conditions [20], [21], the smart contract data are also recorded by executing transactions, and the verified smart contract data are stored alongside the transaction information in a blockchain. Usually, the blockchains that have a virtual machine running on the network also provide built-in programming languages which are used to write smart contracts [3].

1.2.1. Application of Blockchain Technologies

In the recent years, the blockchain popularity has increased as it allows one to perform monetary transactions without relying on third parties. Even without that reliance on smart contracts, blockchain technologies have mostly been applied for asset management purposes [11]. Similarly, most smart contracts also deal with financial operations, such as banking, loan management, and auctions, which might be so merely because the blockchain supports cryptocurrencies out of the gate, thereby making it easier to build solutions for financial operations [7].

The blockchain could be adapted to other domains, as it can be utilised for the decentralization of business processes, which, in turn, could promote trust and remove the need for intermediaries between parties [5]. Other benefits of the blockchain application include [15]:

- Reduced settlement times compared to the regular contracts; this smart contract usage in a domain where agreements need to be settled can provide a faster settlement time because additional intermediate parties are not required.
- Asset provenance and auditability – where applicable – can be achieved by providing the full history of specific assets since the history is public and all the operations, transfers, and ownerships can be easily traced back to the origin.
- Native asset creation supports the recording or creation of the asset ownership records and allows employing these records for future transfer use.
- Enablement of new trust methods, since, instead of relying on any central authorities, the verification responsibility falls on to the blockchain network participants.

Additionally, the main way trust is built into the blockchain is through the immutability of the transaction records on the public distributed ledger. The usage of the blockchain is not always applicable to any domain, and some consideration must be taken before deciding to apply a blockchain to a specific domain. Generally, blockchain is beneficial to employ when multiple untrustworthy participants might exist in a network, and some sort of interaction needs to be recorded in which a trusted third-party authority is not available [22].

The domains to which blockchain is applied are quite limited [7], which could be partly attributed to the immaturity of the blockchain technology, and partly because blockchain and smart contract development needs are not extensively supported [23]. So, in order to accommodate new developers who would like to propose their custom solutions, new development methods are needed. The domains that have been receiving notable attention are education, e-government, healthcare, and transportation.

1.2.2. Blockchain technology platforms

Several blockchain platforms exist that can support smart contracts, but the two most common ones are *Ethereum* and *Hyperledger Fabric* [9]. The Ethereum, EOS and Hyperledger Fabric support Turing complete smart contract programming and are tailored for generic solution development. Additionally, Corda and Stellar only support Turing incomplete smart contracts that are tailored for the development of digital currency applications [13]. A general comparison of the blockchain technology platforms, illustrating the execution engine, and supported languages, is presented in **Table 1**.

Table 1. Comparison of Blockchain Technology Platforms

Platform	Ethereum	Hyperledger Fabric	Corda	Stellar	EOS
Execution environment	EVM	Docker	JVM	Docker	eWASM
Type	Public	Private	Private	Consortium	Public
Smart Contract	Turing complete	Turing complete	Turing incomplete	Turing incomplete	Turing complete
Supported Programming Languages	Solidity, Vyper, Yul	Go, JavaScript, Java	Java, Kotlin	Python, JavaScript, Go, PHP	C++

Of the overviewed platforms which support smart contract development, the Hyperledger Fabric and Ethereum platforms have reached a higher level of maturity [24]. Additionally, the platforms differ by type and are well established in the industry as development platforms with a wider support for developers [25]; based on this, it was decided to analyse these platforms in more detail.

1.2.2.1. Ethereum

Ethereum is a generalised technology that provides a decentralized open-source blockchain with support for executing smart contracts [26] [27]. Ethereum provides and utilises a virtual machine for smart contract execution, which works on top of a blockchain ledger. Any network participant who participates in the network has a copy of the public ledger, on top of which, an Ethereum virtual machine is hosted which is capable of executing and computing the bytecode. Essentially, the Ethereum blockchain provides the ability to maintain an internal database, execute the code, and communicate with other participants. The main advantage of the Ethereum blockchain is the support for developers to write custom smart contracts [20]. Ethereum in the Ethereum blockchain uses *Ether* as *de facto* cryptocurrency, and all the transactions that are done on the blockchain are using this cryptocurrency; additionally, any rewards which the miners receive for their work during the consensus algorithms are also rewarded by using Ether. It should be noted that, in recent years, the Ethereum blockchain started a transition from the proof of work to the proof of stake consensus algorithm.

While the Ethereum virtual machine executes a compiled bytecode, it still supports the development of custom smart contracts in several domain-specific languages, such as Solidity and Vyper, Yul and so on.

Solidity is an object-oriented programming language for smart contract development, it is also the most popular one, and it receives the most support [26]. While Solidity can be utilised in numerous blockchains, the most prominent example is the utilisation of such smart contracts on the Ethereum blockchain. Since the Solidity smart contract code is hosted on the blockchain similarly to the transaction information, once the smart contract code is deployed onto the blockchain, it cannot be changed easily. This complicates the development since there is no way to update smart contracts without losing data. In its stead, new smart contracts need to be developed; unfortunately, it is nearly impossible to pertain or transfer data between smart contracts if this sort of functionality had not been planned in retrospect.

Vyper is another programming language that is supported on the Ethereum blockchain [26]. It is a strongly typed programming language for writing smart contracts similar in syntax to *Python*. Vyper, compared to Solidity, lacks the support of some specific constructs, such as modifiers, inheritance, recursive calls, and so on, and it forgoes these mechanisms for the sake of security and does not aim to replace the Solidity programming language.

1.2.2.2. Hyperledger

Hyperledger is an umbrella term for a project by the *Linux Foundation* that proposes a relatively large number of blockchains for different domains [4]. While Ethereum is the most popular platform for the development of smart contracts in the public blockchain domain, Hyperledger Fabric is a most popular and most supported platform out of private blockchains or projects from Linux Foundation. The Hyperledger Fabric itself is built by using the *Go* programming language which supports CouchDB and several components for the management of organizations

and permissions. It also allows building custom smart contracts by using such programming languages as JavaScript, Java, and Go [13]. Out of the three supported programming languages, Go is the most popular is for building and writing smart contracts.

The *Go* language is the first and foremost in this list because Hyperledger Fabric is built by using Go; so, any newly built features become available in the Go programming language [28]. In the context of the Go programming language, Hyperledger Fabric proposes a new concept, called chaincode, which is akin to the smart contract in other blockchain solutions.

The platforms chosen for further analysis in this dissertation are Ethereum coupled with the Solidity programming language and Hyperledger Fabric coupled with the Go programming language. This is mainly because of the popularity of the platforms, but also because such a combination covers the broadest spectrum of concepts – from private to public blockchains, and from the open source to the custom project. These technologies have also been covered in the scientific literature most of all, compared to the other lesser-known blockchain platforms.

1.2.3. Smart Contract Development Standards

As the blockchain use cases are increasing in scope, scale, and complexity, the need for standardized technologies arises. Unfortunately, the blockchain technology lacks maturity in its implementation, and the majority of blockchains do not offer enough support for the developer [29]. Since there are no general standardized methods or development guidelines that could be employed for blockchain solution development, such platform-specific proposals as the Ethereum Improvement Proposal (EIP) aim to remedy this situation [30].

EIP is a community-based project which proposes Ethereum smart contract standards for developing custom solutions on the blockchain, protocol specifications for Ethereum, and so on. The proposed standards can be utilised during the development, and some of the proposed standards provide templates or exemplary implementations that could be used freely, such as *OpenZeppelin* [31].

The standardization of blockchain technologies would benefit both the developer and the users of smart contracts. Unfortunately, these resources are not always available for other platforms, which complicates the development of blockchain technology-based solutions for specific business needs. It is difficult to gather information about the capabilities of blockchain technologies and to find out where these technologies would be applicable and whether the application would be useful for specific business problems [32]. And, in the context of this dissertation, since such resources *do* exist, particular attention needs to be paid to the support of these already existing solutions.

1.2.4. Blockchain technology-based and smart contract-based system

Blockchain enables the development of distributed decentralized software where a portion or the entirety of software components is hosted on a peer-to-peer network and in which data is shared by using the public ledger [26]. The technology provides a mechanism for defining not only agreements using smart contracts but

full applications running on the blockchain network without relying on intermediaries. Provided the chosen blockchain platform is a public blockchain, smart contracts are open source, and all blockchain network participants can view and verify the code of smart contracts. While any system that uses blockchain technology in any capacity can be considered a blockchain technology-based system, a distinction can be made to a subset of this kind of systems which utilise smart contracts for integrating blockchain in a solution development – smart contract-based systems. It is important to note that the development of custom solutions is most often associated with the development of smart contracts, and this in essence means that the majority of blockchain technology-based systems are in fact smart contract-based systems.

Smart contract-based systems support cryptocurrencies from the get-go if smart contracts are hosted on a public blockchain. In addition, transactions are also carried out by using the main cryptocurrency of the platform. For any data record insertion, an agreement between participants must be reached by employing a consensus algorithm. All the system components, such as the front-end, may be hosted on a decentralized network; still, the core software component, the smart contract, is deployed on the blockchain [33]; whereas, for any other component, there are several hosting options. In essence, a smart contract-based system is composed of a front-end application, which may include back-end services which are used to perform more complex business logic, and a smart contract hosted on a blockchain (**Figure 2**). Coincidentally, the development of software artefacts that will be hosted on the blockchain or decentralized network need to be thoroughly examined and verified, as the development needs to be completed in full and thoroughly tested before the deployment to the blockchain. Such a restriction is particularly evident in the development for smart contracts, as it without any exceptions will be hosted on the blockchain. Additionally, as is often the case, if the system under development decentralizes only a portion of business logic, additional determinations need to be made for domain data relocation to the blockchain, specifically, the smart contract.

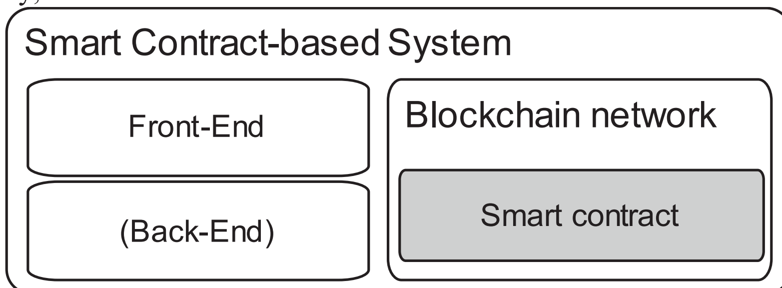


Figure 2. Smart contract-based System composition

It is worth noting that there is an even more generalized term *Decentralized Application* (dApp) which often refers to software applications that, rather than employing strictly a blockchain, can still be hosted on any distributed ledger technology [34]. Often, the DApps are confused with blockchain technology-based, or smart contract-based, software applications since the terms DLT and blockchain

are often used interchangeably. Decentralized applications relate not only to blockchain technologies, but also to distributed ledger technologies (DLTs) in broader terms [35].

Smart contract-based systems are a specific subset of blockchain technology-based systems which, in turn, are considered to be a subset of decentralized applications (dApps). Although the distinction between the three terms can be made, the difference in the academic literature is not as apparent. As a result, moving forward, in the scope of this dissertation, the terms *blockchain technology-based system* and *smart contract-based system* are used. The blockchain technology-based system is considered to be a software application using the blockchain technology in general, and, in a smart contract-based system, at least a part of the business logic is implemented by using a smart contract which is hosted on the blockchain network. Although there are some variations of deployment, this is the main requirement for defining a system based on the blockchain technology.

1.3. Smart contract-based System development

In the area of information systems, the application of the blockchain technology is still fairly limited [3]. To increase the scope of application to other domains, particular attention must be paid to the development of smart contracts, as it serves as the backbone of the blockchain-based software [26]. Currently, smart contract development resembles both the iterative incremental (**Figure 3**, b) and the waterfall development method models (**Figure 3**, a), as the developed software, strictly smart contract, cannot be updated once deployed, while the other components of the system components and the smart contract before deployment can be developed by following continuous or incremental iterative methods (**Figure 3**, c) [36] [13]. The development of smart contract and smart contract-based systems is considered to be in its infancy [26], and generic development guidelines [37] as well as tools to this facilitate development are still required. Currently, the development phases are not as clearly defined as in the traditional software development methods; sometimes they are skipped or combined with the other development phases, or else they do not have any clearly outlined outcomes. Furthermore, the development phases are not as interconnected as they should be, thus allowing the developer to fill the gaps with experience, or with additional analysis during each step [38].

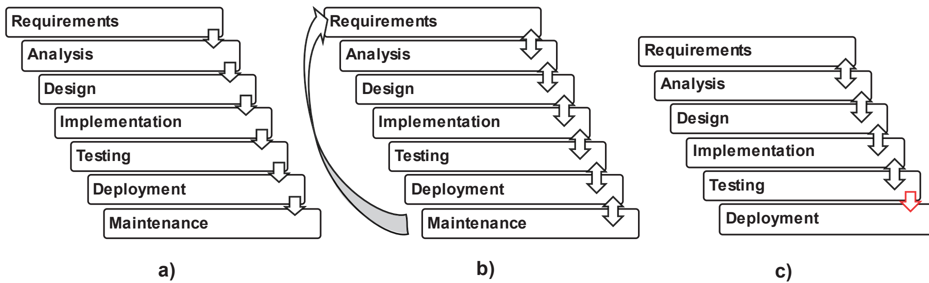


Figure 3. Software development processes: a)Waterfall, b)Continuous deployment, and c) Smart Contract

Requirements. This phase is often combined with the analysis phase. However, the requirements phase in the context of the smart contract-based system development aims to outline and identify the relevant use cases where the blockchain technology would be beneficial, as the technology does not fit every use [39]. Yet, for this objective, the developer also needs clear understanding of the business processes and the blockchain technology. Unfortunately, there is no established common framework, nor does any method exist which could help the developer identify, gather, and develop requirements for smart contract-based systems [26] [13]. Coupled with the fact that the requirement elicitation process is not standardized [39], this means that the developer needs to collect and analyse requirements while being unassisted.

Analysis. During the phase of analysis, the elicited requirements are verified. This step is particularly important as some requirements are not compatible with blockchain in general, and such issues become apparent only after conducting comprehensive analysis [13]. Although, even after thorough analysis, a clear distinction for the applicability of blockchain technologies is still not always clear. Furthermore, there are only a few guidelines that help analyse the already existing blockchain technologies in order to find a suitable platform [22]. Additionally, a comprehensive analysis of the blockchain implementation technologies is performed, as selection plays an important role in any future development [34]. At the end of such analysis, the developer might choose to utilise an analysed blockchain network for the implementation of a system, or they might conclude that the chosen platform is not necessarily applicable to a specific application, and that another blockchain platform thus needs to be analysed.

Design. Once a specific implementation platform has been selected, the design can be started. Unfortunately, the design is not a clearly pronounced phase in the overall smart contract development; in the majority of cases, the design is skipped or started together with the implementation instead. Still, key architectural design decisions need to be made like in a regular system development process. Considering that blockchain-based system development mainly deals with what kind of data and business logic should be relocated to the blockchain, specifically, smart contracts, integration with the other system components, is considered as well. Some decisions are highly specific for the selected technology, such as design patterns, libraries and standards considerations, whereas some decisions are generic for any

implementation platforms, such as state management, data structures and so on; unfortunately, such a distinction is not pronounced in the current methods.

Implementation. Once a selection has been finalized and some design decisions have been made, implementation can start. The established technologies feature comprehensive documentation outlining the smart contract implementation principles, the functionalities of smart contracts, APIs, and so on. Additionally, smart contract implementation can also utilise community-developed smart contract implementation standards, thus making the implementation phase the most supported phase. Based on the analysis of the deployed smart contracts on Ethereum, most deployed smart contracts are based on the ERC standards [40]. The introduction of the smart contract standards made code the cloning and template-driven approach prevalent during smart contract development [41].

The usage of smart contract standard templates is not necessarily an issue when the developer wants to develop a standardized solution. Unfortunately, issues start arising once developers wish to develop custom solutions. Then, the standard implementations need to be used 'as is', or extended manually. Unfortunately, manual extension is quite complicated as the developer needs to thoroughly analyse the provided smart contract code; this again takes a considerable amount of time, ultimately extending the development process [40]. If a smart contract utilises a standard template, even a simple modification can complicate the development as the developer needs to modify any related element of the smart contract.

Testing. Following the implementation, before the deployment, the smart contract requires thorough testing approaches [38], since issues and inconsistencies introduced in the course of development become permanent once published. Ensuring the intended behaviour of smart contracts is important to maintain trust in smart contract-based solutions. The main difficulties arise when identifying test case scenarios or testing the integrations between various system components. Considering that the testing phase lacks well-established testing tools and environments for performing analysis, this could be used for validating or checking for interoperability, security, and integration issues. The most prevalent testing is done by using code reviews, and by deploying smart contracts on a local test network, which may also take up a considerable amount of time.

Deployment. During the implementation, developers become aware of the need to address the immutability aspect of the blockchain technology [42]. The fact that, after the deployment, the smart contract cannot be changed anymore is often not clear for new developers as it requires a considerable investment of time to be spent for the smart contract requirement specification and design. Without thorough analysis, it is often too late to rectify issues after the smart contract deployment, since no additional logic to the smart contract can be introduced anymore, and any issues that are still found after the deployment will remain unaddressed.

In conclusion, by employing such a highly specific, but ill-defined and complicated development cycle, the developer may miss important details [43] [38]. Template-driven development and code cloning practices facilitate smart contract development for experienced developers, but the difficulty for newer developers remains high since they are not familiar with the key smart contract development

limitations. A general standardized approach which would cover the definition, specification, and design of the smart contract-based system structure and behaviour could facilitate such processes of system development.

Several research directions proposing to facilitate the development process exist, ranging from introducing new development approaches to supporting a specific development phase. The automation area is analysed in terms of software testing, deployment, and code generation since model-based engineering approaches can cover a variety of such cases and be used to support requirement elicitation, design, and implementation [42]. The capability to automate the software code production by using the model-based approaches, in the context of this dissertation, shall be analysed further as the primary basis for automating the development of smart contract-based systems.

1.4. Modelling in the software development process

Modelling in software engineering serves as a tool for improving the understanding of the systems by using a common language. The principle which defines modelling is an abstraction which serves as a formalized model for software specification and definition [44]. Modelling also aims to improve productivity by simplifying design processes to promote communication between the developers and the specified models during the system design and serve as a basis for the system under implementation, thereby empowering the implementation of software and systems [45].

Over the past few decades, the introduction of general modelling languages has been observed, and modelling in software engineering is a practical and effective method for supporting software design. Modelling can be used to define the requirements of software in a structured and understandable formal notation. Modelling approaches vary widely from simple model utilisation for communication purposes in strictly model-based engineering (though not encompassing model-driven development approaches), to software code production directly from models produced in model-driven development, and everything in between [45].

Model-based software development approaches, such as MBE, MDD, and MDA, emphasise the importance of models in the software development process, although these differ in terms of the modelling utilisation level. In strictly model-based approaches, the produced model is not directly used for code generation purposes, while, in model-driven development, it is the primary artefact for the implementation process [45]. So, instead of using the model to produce a software code in model-based engineering (MBE), the model of the system under development mainly supports requirement engineering, design, and analysis [46]. In model-driven development, the model is used for the production of the software code. MDA is a framework proposed by OMG, and it relies heavily on other standards by OMG. It provides specific guidelines for the design used to develop software system modelling approaches emphasising model transformations between different abstraction layers [47].

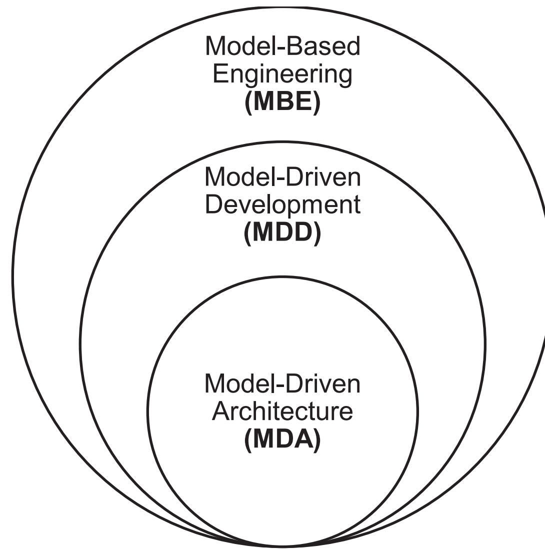


Figure 4. Relationship among model-based approaches [45]

The role of modelling in software development varies between approaches. The common modelling characteristics are overviewed in further sections to outline the key concepts that are the most prevalent and can be incorporated in the development of a modelling proposal.

1.4.1. Modelling Languages

The modelling language is the main tool utilised in any model-based or model-driven development approach. Most modelling approaches applied to software engineering utilise modelling languages in order to describe software systems in a more abstract, generic language, rather than strictly communicating by using the documented, commented code [46]. The modelling language should be analysed thoroughly to determine which languages could serve as a tool for outlining the structural and behavioural features of the software, especially the blockchain technology. In the upcoming sections, the modelling languages that are most commonly used in the model-based software development approaches shall be analysed.

1.4.1.1. Unified Modeling Language

Unified Modeling Language (UML) is a standardized general-purpose modelling language in the domain of object-oriented software engineering [48] and one of the most widely used graphical notations in model-based engineering approaches for specifying software. UML enables the specification of the structure and behaviour of the software.

Since UML is closely related to the object-oriented paradigm, it serves as a great tool for defining conceptual models at a higher abstraction level than the software code, thus enabling the developers to better communicate with each other. UML allows two different aspects – structural, where static software elements can

be represented, and their features, such as attributes, properties, operations, and relationships can be specified; and behavioural, where collaborations among objects of the system elements can be represented.

Although UML outlines 14 types of diagrams, not all diagrams are used equally commonly [49]. Of all the structure diagrams, the class diagram is the most common [45]. It allows specifying classes and their structure by outlining the features, constraints, defining relationships between classes, etc. Meanwhile, composite structures, object diagrams are mostly not used, i.e. they are used very rarely. Other structural diagrams, such as component, package, and deployment diagrams, are used comparatively often, mostly for describing the system architecture. Lastly, the profile diagram allows defining custom stereotypes and constraints, and it is used for outlining the extensions developed for UML.

Beside the static view, UML also allows one to specify the behaviour of elements outlined in the structure diagrams. Out of the behaviour diagrams, the activity diagram is the most common one, closely followed by the use case diagram. The activity diagram allows us to specify control flow and object flow models. State machine and sequence diagrams are other common diagrams for specifying behaviour [45], while other interaction diagrams, in addition to sequences, are rarely utilised [49].

In practice, the use case diagram can be used in combination with other behaviour diagrams; in the use case driven approaches, the use case diagram is used to specify use cases, which can be further detailed by using activity, state machine, sequence, or other interaction diagrams. This, in turn, allows specifying the behaviour of a system under development by outlining not only the requirements but also the behaviour, and the responsibilities of specific system components.

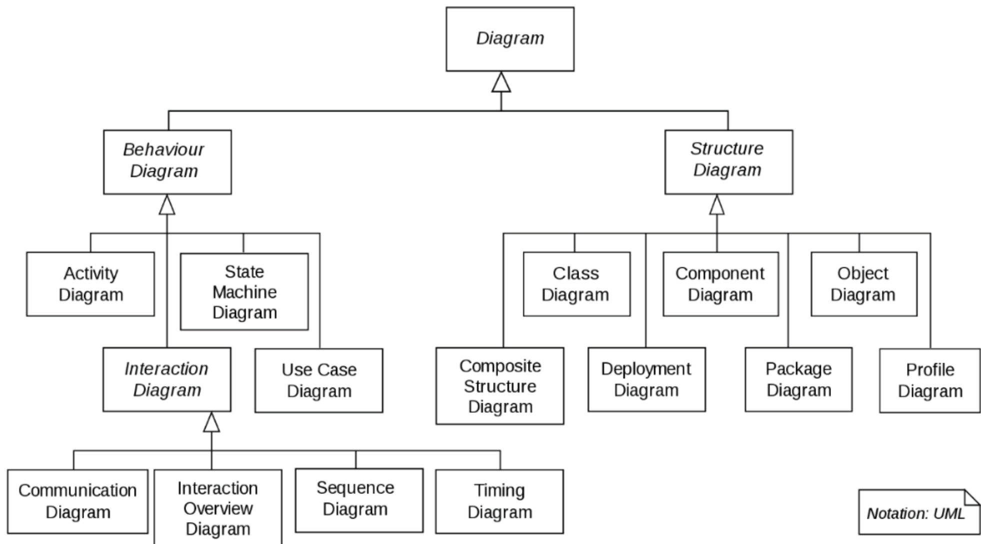


Figure 5. UML 2.5 diagram types [48]

Lastly, when considering UML, additional consideration could be made regarding Foundational UML (fUML) and Executable UML (xUML) extensions to the Unified Modeling Language (UML) which aim to provide more precise and executable semantics for UML models. However, they also have their own limitations when compared to UML. Unfortunately, the current support for fUML and xUML is limited, and only a handful of tools support it. Additionally, based on the fact that xUML introduces additional elements that are not entirely compliant with the UML standard and cannot be mapped directly to UML, it introduces unnecessary ambiguity to modelling. Considering that UML is a well-established standard, the main advantages, such as model execution, simulation, and validation that both of these standard provide can already be implemented more easily in readily available tools.

1.4.1.2. Business Process Model and Notation

The Business Process Model and Notation (BPMN) is a graphical representation to specify business processes in a business process model [50]. BPMN is mainly used to describe the behaviour of business processes at a higher abstraction level. The main goal of BPMN is to provide intuitive notation (for the reader) to represent complex business process details [51].

BPMN is the most common notation for specifying business processes, and it outlines the responsibilities of each business process user, etc. Unfortunately, BPMN does not allow for the specification of the structure, as this is outside the scope of BPMN by design. And, as structure specification is often required when developing software, BPMN is not really suitable for software development purposes. Still, it is a great tool for describing conceptual models at the business

process level, and it can serve as a *de facto* notation which could be understood by many of the stakeholders.

1.4.1.3. Systems Modeling Language

The Systems Modeling Language (SysML) is a general-purpose modelling language for systems engineering applications [52]. While SysML is an extension of UML, compared to UML, it is designed to allow the specification of software and hardware components, rather than the more software-centric design. Similarly to UML, it supports the specification, analysis, design, verification, and validation of complex systems.

Out of nine SysML diagram types, the requirement diagram and parametric diagrams are newly introduced and are often employed for the specification of requirements or block behaviour simulations in academic studies. Other diagrams, such as activity, state machine, block definition, and internal block diagrams, are also fairly popular software engineering activities [53].

Overall, the UML notation is an appropriate way of describing the behaviour and structure of smart contract-based systems, and the use of UML can combine several stages of development in the same model, thus maintaining interoperability and traceability between them. BPMN can be used to model processes realized by the blockchain technology. BPMN basically provides the extended functionality allowing the expression of more complex semantics compared to the UML activity diagram. SysML and UML offer similar tools for model specification, and, if SysML specific diagrams are not used, UML is entirely sufficient for the software structure and behaviour specification.

1.4.2. Modelling Language Extension Mechanisms

Modelling languages can be utilised 'as is', yet, any OMG proposed modelling language includes an extension capability for proposing a new notation or tailoring the already existing general languages to specific domains. This extension can be achieved by a number of mechanisms, ranging from metamodeling to the development of a profile or an entirely new domain-specific modelling language.

1.4.2.1. Metamodeling

Metamodeling is often referred to as the analysis, and development of rules, constraints, and theories for describing and formalizing another model. A *metamodel* is a model of a model, and metamodels are used to define relationships between concepts, such as the model that captures the abstract syntax of a language [54]. Creating a metamodel allows one to outline specific concepts which could then be used to support the analysis of languages for checking conformance, and to support transformation. It should be noted that any OMG proposed standard that supports extensions also conforms to the *MetaObject Facility Specification* (MOF) [55], which is the foundational standard for outlining a common approach on how models can be shared between applications, stored, exported, or imported by using an XML-based standard format.

The main usage of metamodeling enables the outline of mappings between metamodels which are used to outline transformations between specified models in a specific notation to another model conforming to different metamodels [54]. When discussing metamodels, a clear distinction of the metamodel hierarchy should be kept in mind. MOF specification provides guidelines on how to specify metamodel elements which define new domain-specific language elements or allow one to extend modelling languages. Following the guidelines, the hierarchy can be explained as follows: the proposed metamodel defines modelling languages (**Figure 6**, M3); the modelling language defines language-specific concepts (**Figure 6**, M2); using the defined modelling language the user can define a domain model and assign semantic meaning (**Figure 6**, M1); and the last level defines the specific instances of such model elements (**Figure 6**, M0).

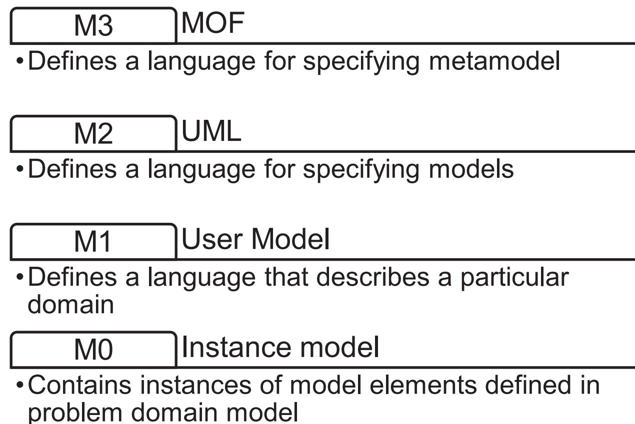


Figure 6. MOF Model Hierarchy

Additionally, XML Meta-data Interchange (XMI) [56] is used as a common format for model-sharing purposes. The main purpose of XMI is to specify how XML tags should be used to outline the MOF model in the XML format. Since XMI is based on XML, metadata tags and descriptions can be contained in the same file. XMI is used not only for models defined in UML, but also for any model based on MOF, regardless of whether these models use custom or extended notations. XMI facilitates the exchange of metadata between different UML modelling tools [56].

1.4.2.2. Extension using profile

Another way of tailoring generic modelling languages is to provide a modelling language extension in a form of a profile. *Unified modelling language* (UML) profiles provide an extension mechanism to customize UML models for specific domains and platforms without changing the structure of the underlying modelling language structure (metamodel) [57]. This extension mechanism enables additional customization of standardized semantics in a strictly additive way so that the extension adheres to standard notation semantics. Profiles are defined by stereotypes, tag definitions, and constraints applied to specific modelling elements. The developer can define new stereotypes, tags, and extensions that can be applied

to UML metaclass elements. Profiles are collections of extensions that tailor UML for specific domains and individually customize them [48].

1.4.2.3. Domain-specific language

The most customizable approach for the definition of a new domain language is to enable the development of an entirely new notation in a domain-specific language (DSL) form [58]. DSL proposes a new specific purpose language that may be applied to a number of models, and to a number of different scenarios. A new metamodel needs to be outlined which would not necessarily adhere to the MOF specifications and which could exist outside the OMG standard ecosystem. DSLs are usually tailored only to a specific domain or platform, even though a distinction between the general language and the domain-specific language is not always clear since DSL may have specific features for a particular domain but can still be applied broadly.

The best extension mechanism is the one that requires the least investment. As long as a general purpose language can be tailored for a specific purpose, it should be, as it does not require stakeholders to learn a new notation, and thus they would only need to analyse the extension details. DSL should be considered last, as the limited applicability makes the design and support costly for any associated tools; on top of that, the developer would need to learn a new notation. Additionally, it is possible to ensure compliance with other languages, especially when model transformations are considered.

1.4.3. Model Validation

Model validation is a task for the verification of models described in modelling languages [54]. In model-based engineering, it is used to check the model for correctness, consistency, and quality issues during and after specification, especially before the implementation starts.

There are two approaches to validation: simulation and verification. Simulation, as the name implies, is often used to simulate the behaviour of the dynamic view of a system. Meanwhile, verification allows us to develop rules for model checking. The most common way to describe such verification rules is the *Object Constraint Language* (OCL). It allows us to develop constraints which could be used as a precondition, postconditions, and invariants to be applied to specific model elements. OCL is also most compatible with UML. By using OCL, developers can write custom validation rules which include evaluation of the model in terms of correctness, completeness, and assess conformance to the specified metamodel, cover test cases, and anything in between.

1.4.4. Model Transformation

Model-to-model transformation is a key aspect of model-driven development (MDD). In model-driven development, the process of automating the production of new models and updating the already existing ones is done by using model transformation [59]. Automatic model transformations can save time and produce fewer errors by avoiding human errors introduced during manual model

transformation processes. Model transformations can be performed in various ways; still, each model transformation includes its own set of expressions, inputs, and outputs. The models used in model transformations are usually defined by specific metamodels, and, during the automatic transformation, adherence of the newly created model to the targeted metamodel is ensured. Model transformation could also help transition between different abstraction layers when specific concepts of specific technology platforms are introduced. By utilising transformations, a generic model can be transformed into a highly specific one.

Two specific types of transformation can be performed in the model-driven development process: model-to-model transformations, or model-to-text transformations. A standard proposed by OMG, QVT stands for query, view, and transformation, which defines three related transformation languages for model-to-model transformations: Relations, Operational mappings, and Core [60]. The ATL language, based on the MOF QVT standard, is a transformation language which allows one to develop M2M transformation rules-based algorithms. The language of ATL transformations allows us to define 3 types of transformation components: ATL transformation modules, ATL queries, and ATL libraries. In addition, these specific pieces can be supplemented with additional functions, attributes, and validation rules [61]. The MOF QVT standard acts as a basis for ATL implementation to support model transformation activities.

Whereas OMG's QVT is used for expressing M2M transformations, MOFM2T is used for expressing M2T transformations. MOF *Model-to-Text Transformation Language* (MOFM2T) is another OMG standardized model transformation language [62]. MOFM2T is part of the Model Driven Architecture and, as the abbreviation implies, it conforms to MOF concepts. It can be used to develop transformations which transform a model into a text (M2T) and which in model-driven architecture is the most common way of producing a software code [59]. MOFM2T can also be used to transform a model into documentation.

QVT, QVT implementation ATL can be used for model-to-model transformations, while the MOFM2T can be employed for model transformations to text and can be employed for documentation, most importantly, for code generation activities.

1.4.4.1. Code Generation

Code generation is the key activity of any model-driven development approach. Although it can be achieved in numerous ways, it allows the utilisation of developed models for the production of software artefacts [47]. As opposed to the manual development of software which could introduce issues during development, automatic code generation can help alleviate these issues.

The main benefit that model-to-text transformations provide is that they improve the development time if system/software models *do* exist. The time needed to be spent writing the software code could be spent developing a model instead, from which, the software code can be produced. Additionally, if transformations are used often enough, and the specific transformations are updated regularly, it could

also allow one to produce a new version of comforting code from the same exact model.

The source code generation in the model-based system development approaches aims at producing the code directly from models. The code production is directly supported by the information that is specified in the model, and, as distinguished in modelling languages, it can be based either on a structural static view of a system, the behavioural view of a system, or on both. Since modelling supports object-oriented approaches, the structural details of a model are employed more frequently for the source code production. Making the code production by using model-to-text transformation is relatively straightforward. The code generation from the behavioural aspects of a model is not as common, as modelling deals with a higher level of abstraction, which not always can be translated straightforwardly into a source code from a behavioural view. In order to support the code generation from a behavioural view, the model needs to strictly adhere to the outlined rules, which requires considerable investment from the transformation developer into the development of the validation rules for the behaviour aspects to ensure that the behaviour is correct before transforming it into a code.

Both structural and behavioural model specifications are used for generating the complete source code. UML is the most popular notation, and, for the structure, the class diagram is the most popular [63]. For behaviour specification, state machines, and sequence diagrams are most often employed, and some developers even choose to embed the code directly by using opaque behaviors [64].

1.5. Modelling in Smart contract-based System Development

Model-based software development approaches utilise modelling at varying levels in the software development process. Smart contract-based system development can benefit from the introduction of modelling techniques, as models can be used to analyse the problem domain, and they can document and visualize the design [65]. The development of smart contracts by using modelling is a relatively new field; hence, unsurprisingly, there is no unified approach [26]. Still, model-based development approaches have become increasingly prominent in the development of smart contracts as they provide systemic design practices. In this regard, analysis of how modelling approaches are integrated into the development of blockchain technology systems shall be presented in the upcoming sections [38] [13].

The analysed approaches are divided into sections based on the utilisation level of models in smart contract-based system development proposals. The sections are divided into MBE, not encompassing MDD; MDD; and, since the most comprehensively defined model-based approach is the Model Driven Architecture, a section for MDA-based approaches.

1.5.1. Model-Based Engineering approaches

During the system development process, modelling in modelling-based engineering approaches is not used directly to produce software artefacts; instead, modelling mainly supports the requirements engineering, design, analysis, and

verification/validation activities. A common approach is for an analyst to create a system model which is then provided to the developer to be used as the basis for manually implementing the software code (automatic generation of the code from the models is not supported in model-based engineering that does not encompass MDD approaches). In this process, models are used more often than not to form a generalized abstraction which is then used to support communication between stakeholders [45].

Similar model-based engineering approaches exist for smart contract-based systems as well, and one of those is a structured development process for decentralized applications [66]. The authors developed an agile development process called ABCDE (*Agile Block Chain DApp Engineering*) which is used to support blockchain DApp development. ABCDE is customized to the Ethereum platform specifically for the Solidity programming language. The authors employ the UML class, and they use case and sequence diagrams to describe the interactions between smart contracts and system actors. The method consists of system goals and user stories specification. Moreover, the use case model and the class diagram development are supported by using UML employing additional stereotypes. And lastly, ABCDE provides a security checklist to assess the security and gas costs.

Another model-based engineering approach is presented in [67]. The authors demonstrate three approaches towards the modelling of a smart contract structure or behaviour: data-driven, structure-driven, and process-driven. For supporting the data-driven approach, the entity-relationship (ER) model is used. While ER is insufficient to design all the aspects of a blockchain-based system, ER can be used to specify the data model. For supporting a structure-driven approach, UML notation is employed, specifically, the UML class diagram. As the name of the approach implies, UML is used to outline the structure of a smart contract, along with the specifying properties and operations. Lastly, for supporting the process-driven approach, BPMN notation is utilised. While BPMN does not support structure specification, it still allows for defining business processes in which system tasks can be outlined, and communications with blockchain can be identified.

A modelling approach for specifying formal ontologies in facilitating the formal specification is presented [68]. The authors extended the already existing ontology for the supply chain management and demonstrated how the ontologies can be employed to outline specific domain concepts and later used for the verification purposes, thus aiding in the development of smart contracts on the Ethereum blockchain platform. The ontologies themselves are specified by using a class diagram, and axioms are defined as constraints.

Another useful tool called *Smart-Graph* is presented in [69]. The tool also utilises UML class diagrams for outlining the structure of deployed Solidity smart contracts, but it uses reverse engineering techniques for the production of such representation.

Model-based engineering approaches support the development by providing the ability to validate models or establish a common framework for stakeholder communication, requirement analysis, and design. Unfortunately, the models are not employed as extensively as in MDD approaches, thus additional analysis of MDD

approaches in which the models are employed to produce the software code is presented in the next section.

1.5.2. Model-Driven Development approaches

Model-Driven Engineering (MDE) is a software development approach that places the focus on the creation and implementation of models for the production of software artefacts [70]. Like in MBE, where models are regarded as abstractions of systems or their parts and are used to design and analyse the principles of the system under development, in the model-driven development, models are additionally directly used for the production of software artefacts to be used in the system development and implementation processes. Although the MBE model exists separately from the system and is used to describe the system at a higher abstraction level and provide principles/guidelines for the software under implementation, the MDD models are used as key artefacts of the development process [45]. In the MDD process, the system implementation artefacts are generated automatically or semi-automatically from models.

Compared to MBE, MDD is more heavily explored in the context of the smart contract-based system development. The main takeaway is that the introduction of a fully supported model-driven approach can support smart contracts, not only because of the provided abstractions which facilitate the understanding of software, but it also can help to introduce validation and verification before the start of implementation [66] [71], or before simulating behaviour, or they can even generate a software code [72]. The application of MDD techniques for the smart contract code generation is explored further in this section.

A domain-specific language called iContractML is defined in [73]. While DSL proposes an entirely new notation, it allows one to specify a smart contract structure. The continuation of this research is also presented in [74] which extends iContractML to support the modelling of behaviour by introducing templates for the common smart contract functions. The specified smart contract structure using M2T (Acceleo MTL) transformations is used to produce Ethereum, Hyperledger Composer, DAML and Microsoft Azure platform smart contracts.

Another DSL called DasContract is presented in [75] and [76]. DSL supports the definition of the non-fungible token smart contract structure, but also by supporting the extended BPMN, it enables the specification of the smart contract behaviour. The smart contract specified in DSL can be used to produce a non-fungible token Solidity smart contract code for the Ethereum platform.

Model-driven engineering techniques supporting Gradle Groovy deployment script generation are presented in [77] and [78]. Particular attention is paid to the UML deployment diagram in which several new stereotypes are introduced. The approach is tailored for the Corda private blockchain. In addition, in [79], the author proposes another approach towards the continuous delivery of blockchain technology-based applications in which deployment configuration files are generated by using model-to-code transformations and used to build automation pipelines or DLT nodes deployment packages.

Since the structure is generated rather straightforwardly, the proposals in many cases tend to focus on the behaviour specification for the code generation purposes.

The MDD approach is presented in [80] which employs a UML state machine diagram for the generation of Solidity smart contract codes in the context of a cyber-physical system. Additionally, the authors used model validations as well, as it is pertinent for the security of CPS. The UML state machine is used to allow the specification of transitions, guards, and support of composite and history states.

Another tool capable of the generation of Solidity smart contracts *VeriSolid* is presented in [81]. The tool supports the specification of transition systems which consist of states and transitions between them. It additionally supports the specification of smart contract variables, expressions, and events for the smart contract behaviour, and it even incorporates an additional state for denoting the transitional state between states.

One more tool for generating the Solidity smart contract code is presented in [82], and it is an extension of the previously developed EFSM tool [83]. In both cases, finite state machines are generated from LTL traces for describing the behaviour of smart contracts, and only the usage slightly varies as, in the latter case, it is used for the testing purposes, while in the former case, it is used to produce the Solidity code.

A Petri Net-based behaviour specification of Solidity smart contracts is presented in [84]. The *Petri Net Markup* language is used not only for smart contract code production purposes, but also for simulating and formally verifying smart contract behaviour. The framework includes graphical modelling based on the Petri Net. Additionally, the Petri Net simulation engine is used for validation and verification, which ultimately allows one to generate the Solidity smart contract code. The proposed approach validation allows for avoiding deadlocks in scenarios, as well as logic issues in the defined business processes. Additionally, the authors published another paper on an extension of the ideas presented in their previous work [84]. The Petri Net-based Workflow-Driven Security Framework [85] still focuses on the security and support of business process modelling, but this time it generates smart contracts for the Hyperledger Fabric blockchain.

Another approach towards generating Go programming language smart contracts is presented in [86]. The approach employs the Web Ontology Language (OWL) ontology specifications and the Semantic Web Rule Language (SWRL) rules for the specification of transaction-focused systems.

The BPMN-based platform for business process and asset management is presented in [87]. The authors extend BPMN with the smart contract and contract invocation elements for specifying business process interactions with smart contracts. The smart contracts integrate the token standards. The tool is able to generate Solidity smart contracts. Another BMPN execution engine called *Caterpillar* with its support for Ethereum smart contracts is presented in [88]. The tool provides a generic workflow handler instead of enabling blockchain-based software or smart contract design. Still, it allows specifying business processes and successfully registering the runtime and workflow records while using blockchain.

A general comparison and overview of the analysed approaches based on the utilisation of structure modelling is presented in **Table 2**. Similarly, a comparison of the behavioural aspect specification of the model-based approaches is presented in

Table 3. The main goal of these comparisons is to determine the prevalent structural and behaviour view representations that deal with the smart contract-based system development. The comparison is made in order to identify the modelling techniques and utilisation beneficial for the code generation purposes. The structural view and the behavioural view are analysed with the consideration of the code production activities with the objective to determine the composition of the defined models, to outline what kind of notations are employed, and what kind of platforms the proposals are tailored to. The analysis is also performed to determine any supplementary activities that the future proposals should support.

Table 2. MBE approaches for smart contract-based system development. Structural view

	[89] [90]	[69]	[67]	[68]	[77] [78] [79]	[66]	[87] [88]	[75] [76]	[73] [74]	[86]
Approach	MDD	MBE	MBE	MDD	MDD	MBE	MDD	MBE	MDD	MBE
Purpose	Code generation	Visualization Communication	Software Engineering Specification	Specification Code generation	Transformation Validation Code generation	Specification	Registry Modelling	Specification	Specification Code generation	Specification
Employed Notation	UML OCL	UML	UML	UML	UML	UML	UML BPMN DEMO	UML	DSL	OWL SWRL
Diagrams	Class diagram	Class diagram	Class diagram	Class diagram	Class diagram Deployment diagram	Class diagram	Class diagram	Class diagram	Custom	Class diagram
Platform	Hyperledger Fabric Ethereum	Ethereum	Ethereum	Ethereum	Corda	Ethereum	Ethereum	Ethereum	Hyperledger Fabric Ethereum Microsoft Azure	Hyperledger Fabric
Language	Solidity Java	Solidity	Solidity	Solidity	Gradle Groovy	Solidity	Solidity	Solidity	DAML Solidity JSON	Go
Code generation	Smart contract structure	-	-	Smart contract structure	Deployment script	-	+	-	Smart contract structure	-
Provided Extension	UML profile	UML profile	UML profile	-	UML profile	UML profile	UML profile	Metamodeling	DSL	-

For specifying the smart contract structure, the UML class diagram and the UML profile are used as a combination as the most common approach. While not all proposals deal with the smart contract code production, most *do* target the Ethereum platform, specifically, the Solidity programming language. There are some proposals that support multiple platforms, where Hyperledger is the next most considered one.

Table 3. MBE approaches for smart contract-based system development. Behavioural view

	[66]	[67]	[87] [88]	[84] [85]	[80]	[75]	[73] [74]	[81]	[82] [83]	[91]
Approach	MBE	MBE	MDD	MDD	MDD	MDD	MDD	MDD	MDD	MDD
Purpose	Specification	Business process modelling	Business process specification	Verification; Code generation	Specification Code generation	Simulation Code generation	Specification Code generation	Verification Code generation	Verification Code generation	Specification Code generation
Employed Notation	UML	BPMN	BPMN	SysML Petri Net	UML	BPMN	DSL	-	-	-
Diagram	Use case diagram Sequence	Process diagram	Process diagram	Activity diagram	State Machine diagram	Process diagram	Custom	FSM BIP model	FSM	FSM
Platform	Ethereum	Ethereum	Ethereum	Ethereum Hyperledger Fabric	Ethereum	Ethereum	Hyperledger Fabric Ethereum Microsoft Azure	Ethereum	Ethereum	Ethereum
Language	Solidity	Solidity	Solidity	Solidity	Solidity	Solidity	DAML Solidity JSON	Solidity	Solidity	Solidity
Code generation	-	-	+	Smart contract boilerplate	+	+	Smart contract structure	+	+	+
Provided Extension	UML profile	-	+	-	UML profile	-	DSL	-	-	-

For modelling the behaviour of smart contracts, the most supported notations are BPMN, followed by UML. The proposals tailored for business process modelling utilise BPMN, while UML is more tailored to be used in combination with the specified structure. The most targeted platform is Ethereum, followed by the Solidity programming language, whereas Hyperledger Fabric is the third most common option. The proposals that support smart contract code production from custom models are mainly based on state machine-like notations. Additionally, given that some model-based proposals utilise sequence diagrams for execution specification, considerations for the support of interaction specification should also be made.

1.5.3. Model-Driven Architecture approaches and their application in Blockchain-based system development

Model-Driven Architecture (MDA) is a system development approach to employ modelling for support requirements elicitation, design, construction, deployment, operation, maintenance, and modification activities during the software development lifecycle [92]. It provides defined guidelines and application principles for building custom model-driven approaches which utilise several abstraction layers. Mainly three different abstraction layers of the *Computation Independent Model* (CIM), *Platform Independent Model* (PIM), and *Platform Specific Model* (PSM) are outlined in MDA (**Figure 7**). CIM describes the system and its context in a business process conceptually, PIM describes a system without a reference to a specific technology, and PSM describes the system in technical details specific to a platform which are necessary for software code production. For the transition between different abstraction levels and code generation activities, MDA underlines the usage model-to-model and model-to-text transformation, respectively. MDA emphasises the use of UML in the development process for the production of implementation artefacts directly from models during the software development process [47] [92]. MDA can be used as a base framework to define the design process of various systems, including smart contract-based systems, essentially distinguishing between different phases of the software development process. Models can be transformed into lower abstraction models, and these models can ultimately be used to generate the software code of implementation artefacts. Most publications on code generation in MDA-based approaches show that the technology can be successfully applied to improve software development [47].

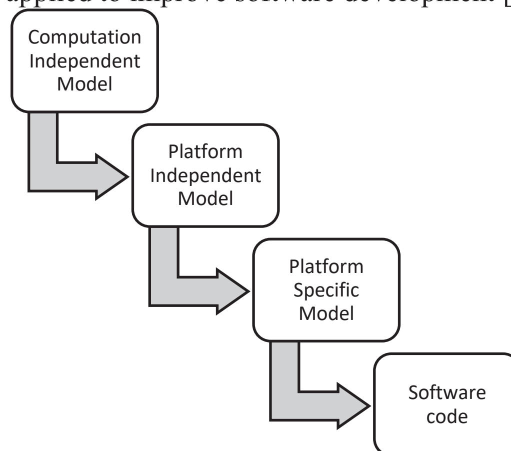


Figure 7. MDA abstraction levels and transitions [92]

MDA-based approaches for smart contract-based system development, or at least approaches that define the Platform Independent Model and the Platform Specific Model abstraction layers [91] [89] [93] [71], are analysed in detail in the upcoming sections.

The above listed sources previously proposed B-MERODE for modelling blockchain-supported business processes [94], whereas, in [93], the MDE4BBIS framework is outlined, which encompasses the PIM and PSM models. PIM is composed of a business logic model and the blockchain technical design model. PSM consists of an intermediary model that, by employing transformations, is used to produce a smart contract code, and it partially covers blockchain connector end-user applications.

The authors of [90] propose an MDA adjacent approach to MDA [89]. The approach defines 3 different layers: the essential layer which is compared to CIM for communication purposes; the infological layer for representing the business exchange compared to PIM; and the datalogical layer which is used to map the model to a specific blockchain platform corresponding to PSM. In any of those layers, a UML class diagram for specification, particularly in PIM, a Smart Contract Ontology is also utilised. The generated code mostly includes the code for structural elements for the Hyperledger Fabric chaincode or Ethereum smart contracts and deployment scripts for the specific platform and does not support the behaviour specifications. The usage of only the UML class diagram limits the generation of a smart contract code.

A systemized approach for the development of smart contracts is presented in [71]. This approach proposes guidelines for the design, verification, and generation of the code and tests. The approach consists of modelling smart contracts by using formal descriptions. The authors propose using model transformations for generating the smart contract code and runtime verifications. The approach could be loosely considered to be based on MDA. It is not entirely clear how smart contracts are modelled as only the verification is presented in more detail. The authors propose to use the event-b modelling language to model the behaviour of smart contracts by outlining axioms. The results of the detailed case studies mainly deal with verification. Thus, other aspects of the approach, such as how model transformations are achieved or how code generation is performed, are not entirely clear.

Another approach based on the MDA framework is presented in [91], in which three different abstraction layers are presented. For CIM, ADICO textual statements are used to start describing attributes, aims, conditions, or, alternatively, components. Out of the three, only the PIM representations are based on modelling, as PIM supports FSMs for modelling behaviour. The specified behaviour is extended with access control, locking, transition counter, and timed transition design patterns during the production of PSM. PSM is considered to be the Solidity smart contract code for the Ethereum platform.

Unfortunately, only [91] and [89] of the aforementioned approaches utilise the CIM abstraction layer or equivalent, and therefore additional analysis of MDA-based approaches is carried out outside the scope of the smart contract-based system development. It shall be presented in the second half of the following section.

1.5.3.1. Computation-Independent Modelling utilisation

A relatively small number of MDA-based approaches using CIM for software development exists, unfortunately, and this number is even lower for the MDA-

based proposals tailored to the blockchain-based system development; therefore, in this section, a broader overview of MDA is outlined. It is also important to note that, in MDA, the usage of CIM is rather limited, and most approaches do not employ this abstraction layer at all, or, if it is employed, it is only used as an intermediate step for producing PIM, rather than strictly a model that outlines business processes and is used for communication only, without outlining any details about the system [95]. Since a relatively limited number of MDA-based approaches for smart contract-based system development exists, a broader scope analysis is presented concerning CIM.

MDA for blockchain proposals, generally like other MDA-based approaches, does not employ CIM [93] [71]. Only source [91] uses ADICO textual statements for describing the smart contract requirements, while [89] proposes the essential layer equivalent to CIM, mainly to be used to outline the ontology which describes the smart contract commitments and asset transfer events.

A comparison of MDA and the adjacent approaches for smart contract-based development is presented in **Table 4**. The main aim of this overview is to determine suitable CIM-level abstractions that can support the development of smart contract-based systems. The comparison is done in terms of the CIM abstraction layer contents, form, existing modelling notation utilisation, and model-to-model transformation implementations. The main takeaway of the comparison is that, currently, the MDA or MDA adjacent approaches support CIM either by having a textual specification, or by employing a class diagram for modelling the structure based on an already established REA ontology. Additionally, the transition between the CIM and PIM abstraction levels is achieved by using manual transformation.

Table 4. CIM utilisation in MDA adjacent approaches for smart contract-based system development

CIM	[71]	[91]	[89]	[93]
Domain	Blockchain	Blockchain	Blockchain	Blockchain
MDA-based	+/-	+	+	+/-
Employs CIM	+/-	+	+	CIM not employed
CIM Modelling Object	Contract parties agreement	ADICO statements	REA ontology	
Employed Notation	Textual	Textual	Class diagram	
Provided Extension	-	-	UML profile	
Transformation implementation	ATL planned	Manual	Manual	
Transformations	Not defined	ADICO to FSM Set of states, Set of transitions, Set of conditions	Not defined	
Output	Not defined	States, transitions, and conditions	Commitment-based ontology	

Output form	Not defined	FSM	UML class diagram	

In model-driven development, outside the scope of smart contract-based systems, CIM represents a business view which is more often than not represented by business process models. While the most popular notation for business process modelling is BPMN [96] [97] [98], UML is also used [96] [97] [99] [100]. The majority of research concerning business processes employ BPMN business processes or collaboration diagrams which are often used as a basis for transformations into use case models [97] [99] and/or classes [97] [99] [101]. This transformation can be performed manually, or the transition can be achieved automatically. Unfortunately, automatic transformations of business process models might limit the expressiveness of business process models since the support for transformations requires the model to strictly adhere to specific rules or modelling approaches [95]. A number of research publications focus on such transformations [96] [99], as OMG does not provide any guidelines for transitions from CIM to PIM.

Additionally, while most of the analysed solutions do not deal with the blockchain as a technology, MDA-based approaches mostly propose to employ CIM to specify functional system requirements [102] [103] [104] [105] [96] [106] [107] [98] [101]. Still, while not every method employs UML, those that do that employ a use case diagram for the representation of high-level requirements [102] [108] [103] [105] [96] or consider the use case model outlined in [97] to be an appropriate CIM level abstraction when there is no more abstract representation available.

CIM is outlined to have a requirement specification which is used then to produce a use case and a conceptual data model [102]. Additionally, the problem domain is described which is used as the basis for determining the system's functional characteristics. The authors of [102] proposed that system requirements can be considered a suitable abstraction for CIM, as these outline the way in which the system/application works without focusing on the structure of the said system. Furthermore, in [108], the authors outline that object-oriented analysis can be used at the CIM level for the use case, class, and behaviour modelling. Use case modelling encompasses the identification and specification of the system functionality from the general knowledge of the domain, and it is usually used as the basis for the system under development.

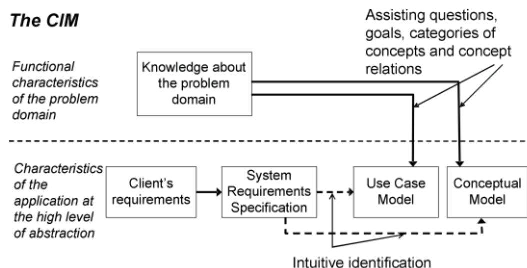


Figure 8. Development of CIM [102] [108]

Sources [103] and [104] present the MDA approach where a business use case diagram is used to specify the functionality and represent the structural view of a

system. The activity diagram is used to describe the behaviour of a system. Additionally, in [105], the approach is extended, and, at the CIM level, SBVR is also used as input supporting the transformations to the use case diagram in the same abstraction layer.

While BPMN and UML are the most popular notations, for CIM specification, several notations can be employed. Source [109] uses data flow diagrams to visually represent the information system data flows from the outside entities and other ways. Even a non-modelling notation can be of use, like in [106], where, at the CIM level, an E3value model is used to represent the requirements and the business value model, or in CommonKADS is used to represent the control structure of the task methods with a pseudocode, and even consideration for the employment of a UML activity diagram as an alternative is made [100].

Similarly, as outlined previously, the comparison of MDAs employing the CIM abstraction layer is presented in **Table 5**. The methods are grouped by author and are compared in terms of the domain for which the method is tailored, the CIM contents and which modelling notations are used to represent them, and whether any extensions to the modelling notations are proposed. Lastly, the supported model-to-model transformations and implementation details are overviewed in terms of the input and output models and specific transformations if outlined in the analysed paper.

Table 5. CIM utilisation in MDA-based methods

CIM	[103] [104] [105]	[107] [98]	[97]	[106]	[100]	[96] [99]	[101]	[109] [95]	[102] [108]
Domain	Software Web applications	Business intelligence	Business processes	E-business information system	Knowledge engineering	Business processes	Business processes	Information system design	System modelling
CIM Modelling Object	UML Business Use Case Model Dynamic view	Requirements SBVR rules	Business processes	E3value model	CommonKADS	Business processes	Business processes	Business processes	Topological Functioning Model Use case model
Employed Notation	UML Use Case Diagram UML Activity diagram	UML Use Case Diagram BPMN business process diagram Textual	UML Activity Diagram BPMN business process diagram	E3 value Model	CommonKADS modelling language	BPMN business process diagram UML Activity diagram	BPMN business process diagram	Data flow diagram Textual notation BPMN business process diagram	UML Class Diagram UML Use case Diagram
Transformation implementation	QVT (MoDAr-WA)	ATL	QVT	ATL	QVT	Semi-automatic ATL	ATL	Manual XSLT	Graph transformations/Manual
Transformations	Actor2Class DataObject2Class Associations2Associations UseCase2Operation Actor2Actor UsecaseSystem2System Include2Ref	Not elaborated upon	ActivityDiagram2ClassDiagram BPSec2ClassDiagram ActivityDiagram2UseCaseDiagram BPSec2Use	Actor to Actor ValueActivity to Lifeline, message/fragments	Method to activity Input to Input Parameter	Not elaborated upon	Not elaborated upon	Not elaborated upon	Not elaborated upon

CIM	[103] [104] [105]	[107] [98]	[97]	[106]	[100]	[96] [99]	[101]	[109] [95]	[102] [108]
	Extend2Alt DataObject2Lifeline Object Action2Message NodeObject2Lifelin eObject FlowFinal2break		CaseDiagra m						
Output	UML domain class diagram UML Detailed Sequence Diagram	Multidimen sional schema	Secure Analysis Class Diagram Use Cases	Behaviour & Interaction model	Not elaborated upon	Use case model, state model, class model, package model	UML class diagram model	UML use case diagram UML conceptual class diagram	Not elaborated upon
Output form	UML Class Diagram UML Sequence diagram	UML class diagram	UML class diagram UML use cases	UML sequence diagram	UML activity model	UML class diagram UML use cases	UML class diagram	UML use case diagram UML conceptual diagram	Not elaborated upon
Provided Extension	-	UML profile BPMN profile	UML BPMN profile	-	UML profile	-	BPMN profile	-	-

While CIM is an abstraction layer that is often ignored even in MDA-based approaches, when used, CIM is commonly employed to outline business process models, and, more often than not, that CIM level is used to specify the system requirements as well. In some cases, the transition between business processes and system requirements is achieved by using transformations at the same level. Generally, CIM usage is underexplored in the academic literature, but the most common notations of the CIM level are UML, BPMN, and textual description. For model-to-model transformation implementation, QVT and ATL are used most frequently. The CIM level is often defined as a feature model, a requirement model, or as a business process model [95]. Unfortunately, no common approach for outlining the CIM contents exists; most proposals adapt CIM to be an abstraction which is used only as the basis for the development of PIM. Considering the composition of such a CIM abstraction level model, the current MDA-based approaches for blockchain development do not support the business process definition and requirement specification; additionally, the approaches do not take into account the smart contract development activities, during which, developers need to outline the requirements for the overall blockchain based solution, what kind of behaviour should be decentralized, and what part of the data model should be relocated to the smart contract.

1.5.3.2. Platform-Independent Model utilisation

In MDA-based approaches, the PIM abstraction level is usually employed for modelling the software structure and behaviour. Based on the previously overviewed model-driven development approaches for smart contract-based system development, the main purpose of the modelling is to support the code generation capabilities. In the case of platform-independent model utilisation that is not always the case, as some choose to support the design or validation/verification activities. Regardless of the modelling application based on the academic research proposal comparison presented in **Table 2** and **Table 3**, just as in MDD approaches, in the MDA-based approaches, the smart contract structure is described by using the class diagram. While the behaviour specification varies, still, the most common way of specifying the behaviour is the state machine-like notation, which includes FSM, Petri Net, and the transition system adjacent notations.

Based on the analysis of MDA-based approaches for smart contract-based system development, each proposal includes a platform-independent model level abstraction [71] [89] [91] [93]. In [91], the layer consists of the FSM diagram which was manually translated from ADICO statements by outlining the states, transitions, and conditions. FSM is made up of states and transitions, and conditions attached to transitions. On the other hand, [89] outlines an infological layer; this specific layer represents the business exchanges which contain goals, commitments, conditions, and actions. These exchanges are defined as operations on the interface. And, in the MDE4BBIS framework presented in [93], this source outlines a platform-independent model representing the analysis and design of the system. The model itself consists of a business logic model, a process model and an activity model allowing to specify the components, the data structures, and the activities order.

Lastly, [71] outlines PIM to be a design of a system, which does not include details about the implementation platform. Unfortunately, the model contents are not explicitly outlined, but it could still be implied that the contract formal specification in Event-B could be constituted as PIM, as it is later used for model validation.

Similarly, as in CIM, a comparison of MDA-based approaches for smart contract-based development is presented in **Table 6**. The main aim of this overview is to determine suitable representation and modelling techniques for smart contract specification which can support the development of smart contract-based systems. The comparison is done in terms of the PIM contents, form, modelling notation utilisation, and model-to-model transformation implementations.

Table 6. PIM utilisation in MDA adjacent approaches for smart contract-based system development

PIM	[71]	[91]	[89]	[93]
Domain	Blockchain	Blockchain	Blockchain	Blockchain
MDA-based	+/-	+	+	+/-
Employs PIM	+	+	+	+
PIM Modelling Object	Formal Smart Contract Description	Finite State Machine	Commitment-based ontology	Business logic model Process Model Activity Model Blockchain Technical Design Model
Employed Notation	Event-B		UML class diagram	
Provided Extension	Not defined	-	UML profile	UML profile
Transformation implementation	ATL planned	Manual	Manual	Manual
Input	Not defined	FSM	Commitment-based ontology	Business logic model Blockchain Technical Design Model
Output	Not defined	Solidity Code	UML class diagram	Intermediary model

Based on the comparison, a conclusion can be drawn that PIM is supported in a variety of ways, and no common model representations can be identified. At this abstraction level, the class diagram is mostly used to describe a structure of smart

contracts and for the behaviour of the state machine-like notation. Similarly, as in proposals outlined in Section 1.5.2, UML is a fairly popular notation for describing the PIM diagrams. Unlike the model drive development approaches, the aforementioned MDA-based approaches do not support model validation and are only tailored to code generation activities. To achieve model transformations, the ATL language is used, but the most common approach is to perform model transformations manually.

1.5.3.3. Platform-Specific Model Utilisation

Finally, the Platform Specific model is used specifically for model-to-text transformation for the production of a software code. The code generation is explored in more detail in the Model-Driven Development approaches section. Generally, the approaches that specify only structural aspects of smart contracts tend to generate only the boilerplate code, and the proposals which incorporate the behaviour as well produce a code from a behavioural view by using the state machine-like notation.

Although most of the approaches do not outline a platform-specific model, the utilisation and specification of platform-specific elements to be used in the direct production of a smart contract code could be considered as one. Unfortunately, the distinction between several abstraction layer models is not as apparent in other proposals which do not utilise the MDA framework. The only clear distinction that can be drawn is between the utilisation of models, as, in some cases, the model is used for validation/verifications, whereas, in the majority of cases, it is used for code generation.

Based on the analysis of MDA-based approaches for smart contract-based system development, each proposal outlines a platform-specific model [71] [89] [91] [93]. In each case, PSM is considered to be a detailed design of a specific component which implements platform-specific details.

Source [89] outlines a datalogical layer which consists of Class, Operations, Constraint, Enumeration, Parameter, Property, ValueMapping, and PackageImport metaclasses, thereby allowing the specification of the smart contract. The specified model is then used by employing M2T transformations to generate a smart contract code for the Ethereum and Hyperledger Fabric platforms. Furthermore, in [91], PSM is considered to be the Solidity smart contract code in itself. Since [71] deals mainly with the validation, it only proposes smart contract code production as a step in the overall method. This specific step aims to produce a Solidity smart contract code from an ER model. And, lastly, source [93] proposes PSM which consists of intermediary models used to produce the Ethereum, Corda and Hyperledger Fabric smart contract code.

Similarly as in CIM and PIM, a comparison of MDA-based approaches for smart contract-based development is presented in **Table 7**. The main aim of this overview is to determine the modelling techniques for the smart contract code production. The comparison is done in terms of the PSM contents, form, modelling notation utilisation, and model-to-text transformation implementations.

Table 7. PSM utilisation in MDA adjacent approaches for smart contract-based system development

PSM	[71]	[91]	[89]	[93]
Domain	Blockchain	Blockchain	Blockchain	Blockchain
MDA-based	+/-	+	+	+/-
Employs PSM	+/-	+/-	+	+
PSM Modelling Object	Not defined	Solidity smart contract code	Domain ontology	Intermediary model/Finite State Machine
Provided Extension	-	-	UML profile OCL constraints	UML profile
Transformation implementation	Not defined	Manual	Acceleo M2T	Not defined
Transformations	Not defined	ADICO statements to FSM Set of states, Set of transitions, Set of conditions	Not defined	Not defined
Input	ER model	FSM	Commitment-based ontology	Object-Event Table Existence Dependency Graph
Output	Solidity	Solidity	Java Solidity	Not defined
Supported Platforms	Ethereum	Ethereum	Ethereum Hyperledger Fabric	Ethereum Corda Hyperledger Fabric

The most supported platform in the MDA-based approaches is Ethereum, followed by the Hyperledger Fabric. The currently proposed approaches mainly deal with the structural specification of smart contracts; additionally, finite state machines are used to specify the behaviour of smart contracts. Currently, most of the overview MDA-based approaches for the smart contract-based system development do not produce a code automatically, or such activities are not really elaborated further upon. Still, at least one proposal employs model-to-text transformations for the code generation which is implemented by using Acceleo M2T transformations.

1.6. Analysis of related works by Lithuanian researchers

The application of the model-based development techniques for smart contract-based system development is not a research area that is under consideration by Lithuanian researchers. Still, while the combination of the research areas is not present, separately, some areas concerning the blockchain, or model-based system development, are analysed by Lithuanian researchers.

Research [110] dealing with blockchain simulation provides an overview of blockchain simulators. The research conducted a systematic review of Proof-of-Work simulators and compared them in terms of the block propagation time, the average block size, and the stale block rate with real blockchain networks.

The authors of [111] developed a framework for selecting the consensus algorithm on the blockchain. The framework incorporates Multi-Criteria Decision-Making techniques which allow one to identify preferable consensus algorithms for specific blockchain systems or applications based on the throughput, decentralization, incentivization, sustainability, and security criteria.

Source [112] analysed the vulnerabilities of smart contracts by using 3 different tools and explored the excess gas consumption fees on 6 auction solutions. The authors concluded that power auction trading is feasible on the Ethereum blockchain only for a small number of agents, and, if such a number increased, the gas expenditure would increase as well. But, when considering that the Ethereum blockchain is transitioning to a different consensus algorithm, the experiment results may change in the future. Source [113] conducts analysis of some problem areas and highlights when the development of the collective intelligence technological solutions using the Decentralized Autonomous Organization platform is supported by the blockchain. The authors conclude that decision-making systems depend on the reputation of their developers, and that their adoption is limited.

Another avenue of interest for the Lithuanian researchers is the model-driven development. Model-to-model transformations are developed in [114] for supporting the transition between the use case model specified in UML and SBVR business vocabularies. The developed model transformations also incorporated drag-and-drop actions which introduce flexibility during the model transformation process when alternatives need to be considered, or when additional expert decisions need to be made. The developed drag-and-drop actions could potentially be incorporated into a multitude of other model-based approaches.

In this paper, an approach using the MDA-based approach is presented [115]. It incorporated the causal modelling approach by outlining a new abstraction knowledge discovery layer. Such a layer would help to trace causal dependencies between the MDA abstraction layers and shift the focus from the external modelling paradigm to the internal paradigm.

Research [116] describes transformation algorithms for the generation of dynamic models from an enterprise model. Specifically, the research proposes UML behaviour element mappings to the enterprise processes, actors, and business rules to support the transformations to UML sequence diagrams. The main takeaway is that the generated models can support the design stages of knowledge-based system development.

1.7. Analysis summary

1. The current smart contract-based system development resembles a waterfall software development lifecycle. The process of developing smart contract-based systems is highly specific for any given blockchain platform, and, during the preliminary phases of development, a considerable amount of effort is spent

identifying the applicability of the technology for specific requirements. The main issue lies in the fact that the key component of any smart contract-based system, the smart contract, because of the blockchain immutability, cannot be changed once deployed, and this outlines the need to focus the development effort on the earlier stages. This, coupled with the fact that the relevant development guidelines and practices for smart contracts are ill-defined, and that no well-established methods exist for the development of such a system, makes the current smart contract development complicated.

2. The utilisation of modelling differs between the proposed approaches for the model-based blockchain and the smart contract development, but it can be used to support automated development activities. The current model-based proposals facilitate and, in some cases, the automate smart contract-based system or smart contract development activities ranging from the support of requirement specification, definition, validation and system design to the code production from models.
3. Currently, the majority of model-driven development approaches for smart contract-based systems for modelling the behaviour of smart contracts employ state machine or adjacent notations which are often supported by the smart contract structure specifications using class diagrams. The specified models are used for generating the Solidity code for the Ethereum platform, and a significant portion targets the Hyperledger Fabric platform.
4. The analysis of the application of strictly MDA-based approaches for the specification of the blockchain technology artefacts has shown that most of the proposals are in the conceptual stages, and these approaches do not fully employ model-driven development techniques throughout the entirety of the software development lifecycle. MDA approaches for the blockchain and smart contract development do not employ the model to model transformations, nor do they offer any multiplatform support. Actually, they are mainly tailored for the Solidity code generation for the Ethereum platform.

II. THE PROPOSED MDA-BASED METHOD FOR DEVELOPMENT OF SMART CONTRACT-BASED SYSTEMS

To facilitate the development of smart contract-based systems, a method based on MDA principles is presented in this chapter. The method enables the specification of the smart contract-based system structure and behaviour in a widely accepted UML notation. The method covers four stages of development of the blockchain-based system (**Figure 9**). The smart contract is considered the key artefact of the blockchain-based system [33], thus the focus of the method is the specification and generation of smart contracts. For each stage of model development, a UML profile is provided (represented as input), and, by using it, a model is defined and used in the next stage (represented as outputs). UML together with its UML profile extensions for the *Blockchain Computation Independent Model* (CIM), the *Blockchain Platform Independent Model* (PIM), and the *Blockchain Platform Specific Model* (PSM) are to be used as a notation for the modelling structure and behaviour of the smart contract-based system. For clarity, the diagrams that the model definition step covers are represented alongside the method. The development steps are enhanced by automating the transition between the different abstraction layers by using model transformations, which ultimately results in a model that is used to produce the smart contract code for a chosen blockchain platform.

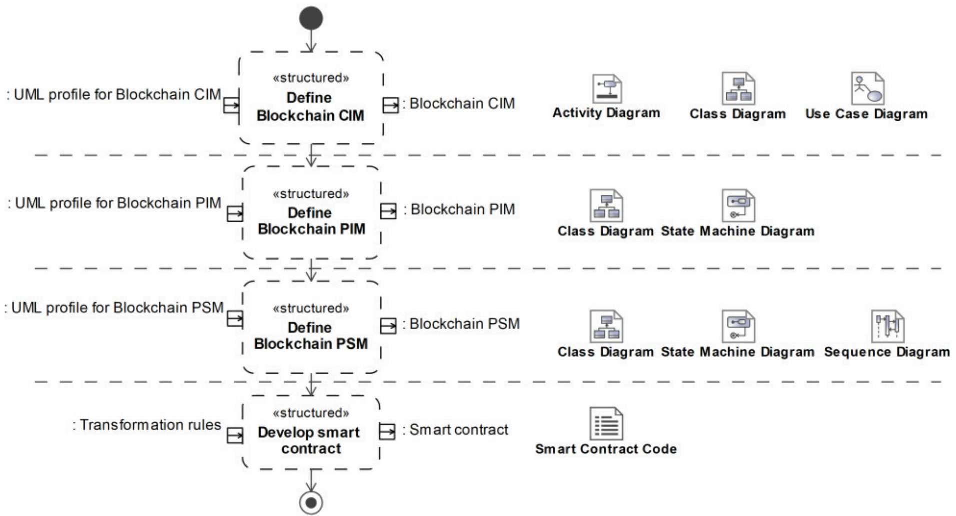


Figure 9. MDA-based method for Development of Smart contract-based Systems

Following MDA [117] [92], the first development stage is the definition of Blockchain CIM. The purpose of this model is to define how the blockchain technology could be integrated into the restructuring, reorganization, and decentralization of specific business processes (represented by the activity diagram), as well as outline the solution requirements as a use case model (a use case diagram) and/or a domain entity model (a class diagram). The use case model represents the

functionality of the system under development – i.e. what the system is expected to do, and outlines the integrations with the blockchain, but hides all the implementation technology-related specifications so that to remain computation-independent.

The Blockchain CIM is then used as a basis for the development of the Blockchain PIM, and, by using model-to-model (M2M) transformations, the Blockchain CIM is transformed into Blockchain PIM. Afterwards, the transformed smart contract structure can be extended by the developer, or the smart contract behaviour can be outlined. Traditionally, PIM represents the design of the system without the details about its implementation. Considering that the proposed method is tailored for the blockchain, the defined models include some specific concepts related to the blockchain technology but are generic enough not to be based on any specific blockchain implementation platform.

After specifying the Blockchain PIM, the definition of Blockchain PSM takes place. As previously outlined, the Blockchain PSM definition is also done by using the previously defined Blockchain PIM model, and once again by using the model transformation techniques. The defined M2M transformation rules specify how models are transformed and are further elaborated in the Blockchain PIM to Blockchain PSM transformation sections. The Blockchain PSMs are defined and tailored to a specific platform, and, in the scope of this research, are demonstrated on the Ethereum and Hyperledger Fabric blockchain implementation platforms. The method enables the definition of the Solidity and Go programming language smart contract structure and behaviour specification.

In the final stage, by applying the model to text transformations, the Blockchain PSM is used to generate the smart contract code of Ethereum and/or Hyperledger Fabric platforms. The method provides the capability to facilitate, and at least partially automate, the smart contract development process by providing a more structured approach for describing smart contract-based systems and thus expanding the potential applications of the blockchain technology.

Since the method is based on the MDA framework, the proposed MDA-based method development stages are elaborated further in this chapter. The Blockchain CIM, Blockchain PIM, and two Blockchain PSMs – Ethereum and Hyperledger Fabric – are tailored for different smart contract-based system development process objectives; therefore, each model's contents and the model development guidelines are outlined in the upcoming sections.

2.1. Blockchain CIM

In MDA, CIM is a tool for the domain expert to communicate the domain knowledge to other stakeholders. So, in the context of this work, the Blockchain CIM serves as a blueprint of what the smart contract-based system should represent regarding its structure and behaviour. For this purpose, the Blockchain CIM encompasses the business process model as a basis for the use case model and the domain entity model development.

2.1.1. Blockchain CIM UML profile

The Blockchain CIM is used to capture business processes, but the focus is placed on the definition of the smart contract-based system requirements which can be used as an information source for supporting automatic transitions between different abstraction layers. Following the MDA, the model specification is done by using UML [92], the Blockchain CIM and any subsequent model notation are entirely based on the UML metamodel [48], additionally extended by the proposed blockchain technology concept stereotypes. As a result, for the business process model definition, the developer can employ the UML Activity diagram, or even use BPMN, a more expressive notation. Nevertheless, the contents of the business process model are neither constrained by any rules of the proposed MDA-based method, nor are the guidelines provided for such development, as the business process model contents fall under the developer's discretion, as no automatic transition between the business process model to the system requirements is defined.

For the use case model specification, the UML Use Case diagram is employed, whereas, for the design of the domain entity model, the UML Class diagram is utilised. The metamodel with extended stereotypes is presented in **Figure 10**. The Use Case diagram and the Class diagram notation are extended by a provided Blockchain CIM profile which includes two specific blockchain technology denoting stereotypes (**Table 8**): «blockchain» and «on-chain». One is used in the Use Case diagram to denote the system integration with the blockchain, while the other is employed in the Class diagram for specifying the domain entities.

Figure 10. Blockchain CIM. Extended UML metamodel

Table 8. Blockchain CIM stereotypes

Stereotype	UML Metaclass	Description
«blockchain»	Actor	This stereotype could be applied to an actor to indicate blockchain integration to the specific use cases
«on-chain»	Class, Property	This stereotype could be applied to data

		class or property that should be relocated to the smart contract

In total, the Blockchain CIM profile contains the outlined stereotypes and additionally contains the Blockchain CIM validation rules (**Figure 11**). While using the outlined profile, Activity, Use Case, and Class diagrams are specified and later validated.

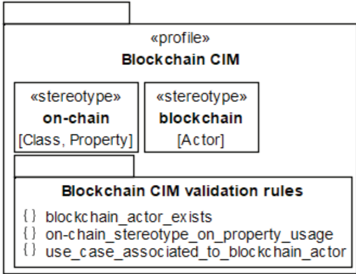


Figure 11. Blockchain CIM Profile

Detailed steps for the Blockchain CIM definition and model validation are presented in the upcoming sections.

2.1.2. Blockchain CIM Definition

The definition of Blockchain CIM starts with the development of the business process model (**Figure 12**). For the purposes of this dissertation, the business process model does not have any specific prerequisites or rules as the transitions from business processes to system requirements are achieved manually by the model developer.

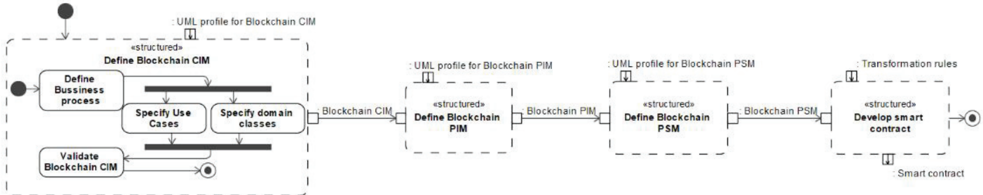


Figure 12. Detailed Blockchain CIM definition step in the context of the method

The requirements for the smart contract-based system are represented by using the Use Case diagram and the domain entity Class diagram. When using the defined business process model, the developer is expected to employ use case modelling to describe functional requirements with detailed interactions between the user, the information system, and the blockchain. Additionally, domain modelling, a practice for representing the domain data entities enabling the developers to identify concepts, relationships, and features in the object-oriented approach, is used for representing domain data entities. The creation of the conceptual data model based on the business processes also falls under the model developer's responsibility and

serves as a basis for the smart contract-based system data model definition. The use cases and the domain entity classes are specified concurrently, as they would be developed in a regular software engineering process.

The use case model consists of Use Cases, Actors, and their relations. Specifically, as outlined in the Blockchain CIM profile, the «blockchain» stereotype is applied to the Actor representing the blockchain ledger. The development of the Use Cases pertinent to the development of the smart contract is analysed below, whereas another Use Case specification is left up to the developer. The only requirement is that at least one Use Case exists, which needs to be implemented by using blockchain, such Use Cases should be associated with the «blockchain» Actor. Additionally, the domain model can be specified, and again, the development of the entities and data structures concerning the smart contract development shall be analysed below. Nevertheless, the Data entities, or Properties that need to be relocated to the blockchain, should be denoted by using the «on-chain» stereotype.

2.1.2.1. Blockchain CIM Validation

The Blockchain CIM profile consists of several validation rules which validate the model in the completeness and correction aspects so that to ensure that the model is ready for the upcoming transformation to the Blockchain PIM. Therefore, the invariant constraints that have OCL specification are outlined as validation rules (**Figure 13**), implementations of which are presented in **Table 9**.

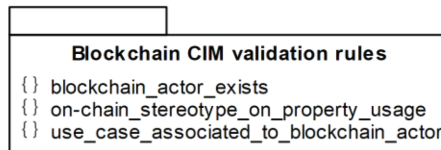


Figure 13. List of Blockchain CIM validation rules

Additionally, during the development, the Blockchain CIM can be validated by using the provided validation rules, or the validation is performed after the use case and the domain entity models have been outlined.

In order to utilise the method, the specified Blockchain CIM should include a «blockchain» actor. This is checked during the validation by the *blockchain_actor_exists* rule. Additionally, if such an Actor is specified, at least a single Use Case associated with the «blockchain» Actor should be identified as well; otherwise, the method would not produce any smart contract artefacts. This rule is validated by using the *use_case_associated_to_blockchain_actor* validation rule. Such determination is made as, during the transformation, based on the specified requirements, a smart contract specification will be created with the relevant operations. Lastly, if outlined in the domain entity model, the domain Classes and their Properties are checked by the *on-chain_stereotype_on_property_usage* rule for the «on-chain» stereotype usage. As the specified domain Classes and Properties using the «on-chain» stereotype will be used to outline smart contract data structures, these will be relocated to the blockchain.

Table 9. Implementation of Blockchain CIM validation rules

<table>
<tr><th>Validation Rule</th></tr>
<tr><td>context Package inv blockchain_actor_exists:
self.ownedElement->exists(e: Element | e.oclIsKindOf(Actor) and
e.appliedStereotypeInstance.name='blockchain')</td></tr>
<tr><td>context Property inv on-chain_stereotype_on_property_usage:
self.appliedStereotypeInstance.name='on-chain' implies
(self.owner.oclAsType(Class).ownedAttribute->exists(at | at.type.oclIsKindOf(Class) and
at.type.appliedStereotypeInstance.name='on-chain'))
or (self.owner.oclIsKindOf(Class) and self.owner.appliedStereotypeInstance.name='on-chain')</td></tr>
<tr><td>context Package inv use_case_associated_to_blockchain_actor: self.ownedElement->exists(a |
a.oclIsKindOf(Association)
and a.oclAsType(Association).endType->exists(ac | ac.appliedStereotypeInstance.name='blockchain'
and ac.oclIsKindOf(Actor))
and a.oclAsType(Association).endType->exists(uc | uc.oclIsKindOf(UseCase)))</td></tr>
</table>

The validated Blockchain CIM is further used as an input for the following MDA-based method step – the definition and transformation to Blockchain PIM. The validation rules could be used extended to support the development of the smart contract-based system specification, rather than strictly validate the conformance to the expected model structure.

2.2. Blockchain PIM

In MDA, a Platform Independent Model (PIM) is used to define the system architecture without the details of a specific implementation platform. In our approach, Blockchain PIM is used to represent the smart contract, the main component [33] of any smart contract-based system, in a generic view independent of the specifics of the implementation platform.

2.2.1. Blockchain PIM UML profile

The Blockchain PIM also uses UML notation, the relevant UML metamodel elements are presented in **Figure 14**. The Blockchain PIM is specified by using the Class Diagram and the State Machine Diagram, while additionally denoting specific blockchain technology concepts using the provided UML profile for the Blockchain PIM stereotypes (**Table 10**). The goal of PIM is to guide the transition between the specification of business processes and the system requirements to a Blockchain PSM, so, in this method, the Blockchain PIM is used to capture the smart contract structural and behaviour details.

Figure 14. Blockchain PIM. Extended UML metamodel

For the smart contract structure definition, a Class diagram is used, where smart contract Operations, their Parameters, and Properties can be defined, and additional structural features are also supported, such as Enumerations and Constraints. Moreover, for behaviour specification, a State Machine diagram is employed, in which, States, Transitions, and guard Constraints can be outlined similarly to the other model-driven smart contract-based system development approaches.

Table 10. Blockchain PIM stereotypes

Stereotype	UML Metaclass	Description
«SmartContract»	Class	This stereotype could be applied to a class to indicate that the class is a smart contract
«pay»	Operation, Property	This stereotype could be applied to a data class or property that should be relocated to the smart contract

The Blockchain PIM is tailored to the smart contract-based system development, and it captures the common structural and behavioural features that can be further refined during the Blockchain PSM development. Just like the Blockchain CIM, the Blockchain PIM has an outlined UML profile which includes three specific concepts of smart contracts: an address datatype, an additional stereotype to specify the «SmartContract» Class and the «pay» stereotype that outlines financial operations (**Figure 15**).

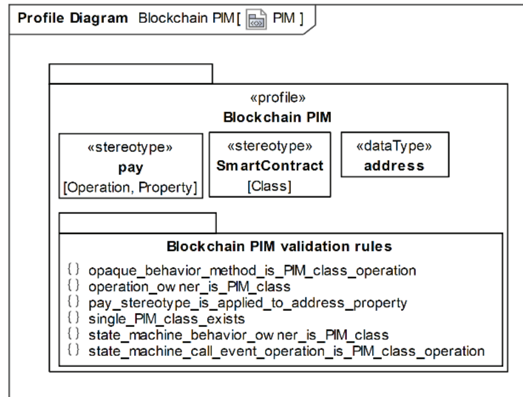


Figure 15. Blockchain PIM Profile

The Blockchain PIM profile composition is further elaborated upon in the upcoming sections.

2.2.2. Blockchain PIM Definition

The Blockchain PIM definition consists of three major steps. During the first step, the Blockchain CIM is transformed into Blockchain PIM by using the model transformation techniques. This transition is automated and achieved by using model-to-model transformations, during which, from the Blockchain CIM, smart contract structural elements are extracted to be included in the smart contract, the transformation is further elaborated in the upcoming section. The M2M transformation implementations are provided in *Appendix 1. Blockchain CIM to PIM Model transformation rules*. Additionally, once transformed, the Blockchain PIM can be extended by the developer, and lastly, the model is validated by using the outlined Blockchain PIM OCL validation rules.

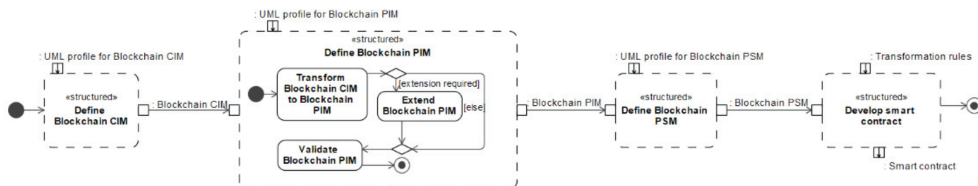


Figure 16. Detailed Blockchain PIM definition step in the context of the method

The blockchain PIM development steps are elaborated further in the upcoming sections.

2.2.2.1. Transformation of Blockchain CIM to Blockchain PIM

The transformation of blockchain CIM to Blockchain PIM starts with the Blockchain PIM «SmartContract» Class creation and the analysis of the Use Cases and the domain model (**Figure 17**). The information specified in these models is relocated to Blockchain PIM. During the transformation, the «SmartContract» Class based on the specified Use Case model is appended to include Operations. These

Operations are created directly from the Use Cases that were associated with the «blockchain» Actor as specified in the Blockchain CIM, relation one use case to one operation. Additionally, provided that a domain entity Class model has been specified, Classes with an «on-chain» stereotype are transformed into a PIM contained Class. Additionally, any Associations between «on-chain» Classes are transformed as well, and Associations between the «on-chain» Class and the regular domain Class are transformed to a Property denoting an Association end as long as the Association end has an «on-chain» stereotype as well.

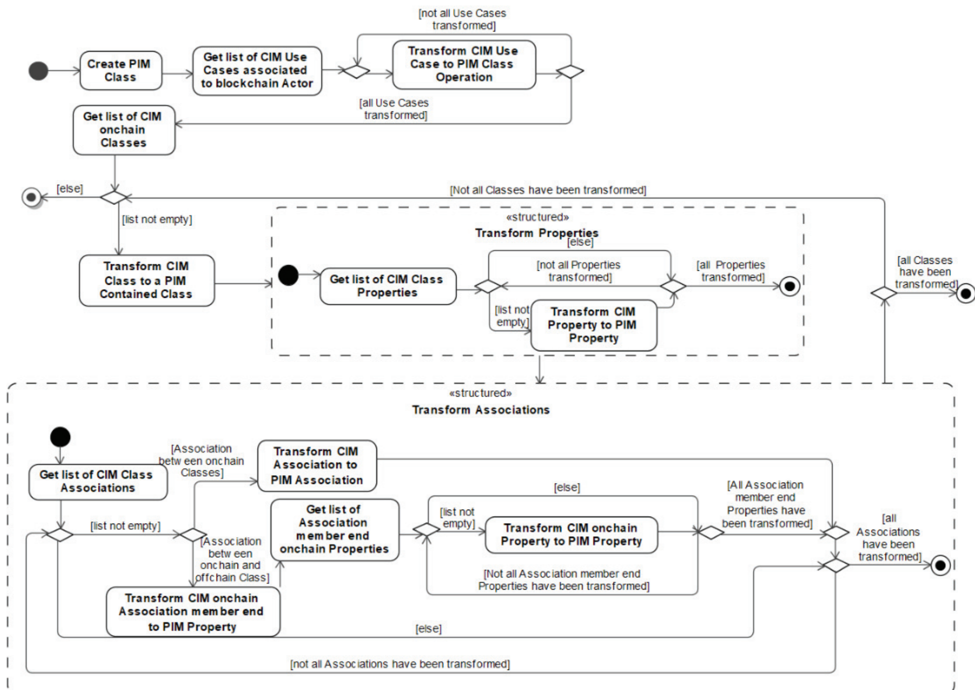


Figure 17. Blockchain CIM to Blockchain PIM Transformation

Similarly, each «on-chain» Class Property is transformed to a PIM containing Class Property. Otherwise, the Properties with the stereotype of «on-chain», which are not contained in the «on-chain» domain Class, are transformed to a newly created «SmartContract» Class contained Class. These attributes are transferred to a Class that was specified by using an «on-chain» stereotype which previously had a multiplicity of 1 denoted on the opposite Association end, the naming of which corresponds to the Class name. At the end of the Blockchain CIM to Blockchain PIM transformation, the developer is presented with a specified «SmartContract» Class with a number of Operations, and, based on the domain model, it could have additional contained Classes (data structures) included as well.

2.2.2.2. Extending the Blockchain PIM Definition

After the transformation, the developer is provided with a class diagram which denotes the Blockchain PIM smart contract structure. The developer can use the model ‘as is’, or may choose to manually extend it with additional details from the specified CIM, like the smart contract structure by adding additional Properties, Operations, and owned Classifiers that define the «SmartContract» data structures or Enumerations.

The developer may then choose to outline the «SmartContract» behaviour specification, which can be done by using the UML State Machine diagram. The State Machine behaviour is based on the previously outlined business processes, and it should use the transformed or additionally defined smart contract Operations. During the State Machine specification, the developer specifies the States for a smart contract, or contained Class, and determines the Operations as Call Events that trigger the Transition. The Transitions can also be specified as Time Events, or have such specified Transition conditions as guard Constraints and/or Effects Opaque Behavior. The Effect corresponds to the broadcasted signal which denotes that a certain event has occurred and should be recorded. The State Machine transformation when stemming from the choice and junction Pseudostates is also supported by the method. The State Machine is specified as an owned Behavior of a «SmartContract» Class, and, since the Class can have multiple Behaviors, which means that multiple State Machines exist in a Blockchain PIM, one State Machine can be outlined as the classifier Behavior, which is relevant during the transformation to Blockchain PSM. But, before the defined Blockchain PIM is used as an input for the transformation to Blockchain PSM, validation of Blockchain PIM occurs.

2.2.2.1. Blockchain PIM Validation

Similarly as in Blockchain CIM, during the validation, Blockchain PIM is checked for correctness and completeness necessary for enabling the model to model transformations. The list of Blockchain PIM validation rules is presented in **Figure 18**, and the implementation is presented in **Table 11**. While the provided validation rules list covers the relevant UML metamodel extensions and validates the conformance to the expected model structure, such a list could be extended with utility validation rules to facilitate the model specification.

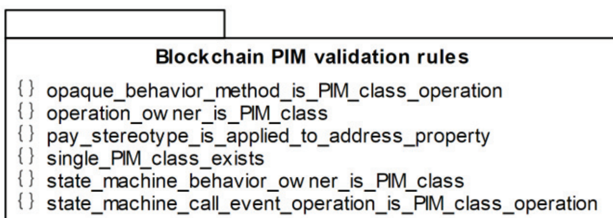


Figure 18. List of Blockchain PIM validation rules

By using the provided Blockchain PIM validation rules, the Blockchain PIM is verified for conformance to the expected smart contract model structure, during which, the model is checked to have a single «SmartContract» Class specified. Any containing classifiers, such as Classes, Enumerations, Operations, State Machine behaviours, and Opaque Behaviors, are contained by the said «SmartContract» Class. Lastly, it is determined whether the specified Opaque Behavior or State Machine Transition Call Events are associated with the PIM «SmartContract» Class Operations. Particularly, the «SmartContract» Class behaviour specification is checked for Operation usage, thereby ensuring that the correct smart contract triggers the transitions through the states. Additionally, the stereotype usage is checked as well, thus determining whether the «pay» stereotype is correctly applied to the smart contract address Property.

Table 11. Blockchain PIM validation rule implementation

Validation Rule
context OpaqueBehavior inv opaque_behavior_method_is_PIM_smartcontract_class_operation: self.specification.oclIsTypeOf(Operation) and self.specification.owner.appliedStereotypeInstance.name='SmartContract'
context Operation inv operation_owner_is_PIM_smartcontract_class: self.owner.appliedStereotypeInstance.name='SmartContract' and self.owner.oclIsTypeOf(Class)
context Property inv pay_stereotype_is_applied_to_address_property: self.appliedStereotypeInstance.name='pay' implies self.type.name='address'
context Model inv single_PIM_smartcontract_class_exists: self.ownedElement->collect(e | e.oclIsKindOf(Class) and e.appliedStereotypeInstance.name='SmartContract')->size()=1
context StateMachine inv state_machine_behavior_owner_is_PIM_smartcontract_class: self.owner.appliedStereotypeInstance.name='SmartContract' and self.owner.oclIsTypeOf(Class)
Context Trigger inv state_machine_call_event_operation_is_PIM_class_smartcontract_operation: self.event.oclIsKindOf(CallEvent) implies (self.event.oclAsType(CallEvent).operation.owner.oclIsKindOf(Class) and self.event.oclAsType(CallEvent).operation.owner.appliedStereotypeInstance.name='SmartContract')

Once validated, Blockchain PIM is further used as an input for the definition and transformation to Blockchain PSM.

2.3. Blockchain PSM

In MDA, the *Platform Specific Model* (PIM) is used to define the artefact details for a particular platform. In the context of the proposed MDA method, the Blockchain PSM defines how the Blockchain PIM should be implemented by using a specific blockchain technology, and a model is specified by using the provided UML profile for Blockchain PSM, which is again an extension of the UML metamodel.

Since the method is tailored to two specific blockchain technology platforms, each section first outlines the general details of the Blockchain PSM, and then, in subsections, the Ethereum PSM and Hyperledger Fabric PSM details are elaborated further upon.

2.3.1. Blockchain PSM UML profile

2.3.1.1. Ethereum PSM UML profile

The Ethereum PSM is used to describe the structure and behaviour of a smart contract intended to be hosted on the Ethereum blockchain. The main challenge of any PSM is to support a relative view with essential details of the specific implementation platform in order to support transformations to the execution code. As a result, since the Ethereum blockchain supports the development of a smart contract in multiple programming languages, and, while considering the fact that Solidity is the most popular option, the Ethereum PSM is tailored to the Solidity programming language.

Similarly, as in the Blockchain PIM, the smart contract structure is specified by using the Class diagram, and the smart contract behaviour can be specified by using the State Machine diagram. In the Ethereum PSM, the State Machine diagram is also used to describe the overall execution of the smart contract States, state Transitions and the extension of the smart contract function behaviour. Additionally, at the Blockchain PSM level, the smart contract behaviour can be specified by using the Interactions, and Opaque Behaviors. The Ethereum PSM profile extensions for the UML metamodel are outlined in **Figure 19**, and the provided stereotypes deal mainly with the specification of smart contract structure (**Table 12**).

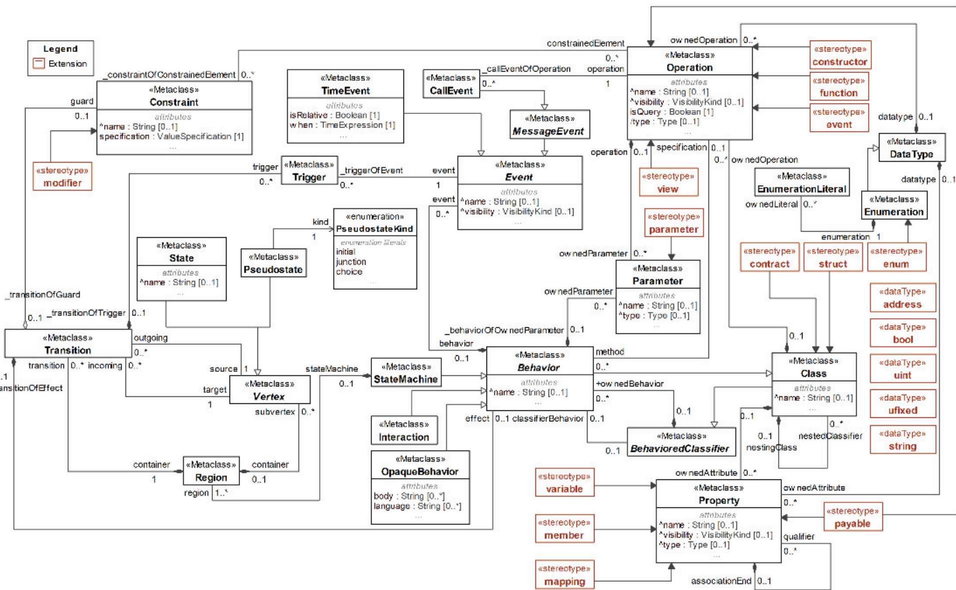


Figure 19. Ethereum PSM. Extended UML metamodel

The Ethereum PSM profile (**Figure 20**) also includes a number of datatypes specific to the Solidity programming language, and it also includes a library of Opaque Behaviors based on smart contract implementations, the Opaque Behaviors would include code embeddings based on the smart contract implementations

provided in the Solidity documentation [118] or the community developed smart contract development standards [30].

Table 12. Ethereum PSM stereotypes

Stereotype	UML Metaclass	Description
«constructor»	Operation	This stereotype could be applied to an operation to indicate that it represents a Solidity constructor.
«contract»	Class	This stereotype could be applied to a class to indicate that it represents a Solidity contract.
«enum»	Enumeration	This stereotype could be applied to an enumeration to indicate that it represents a Solidity enum.
«event»	Operation	This stereotype could be applied to an operation to indicate that it represents a Solidity event.
«function»	Operation	This stereotype could be applied to an operation to indicate that it represents a Solidity function.
«mapping»	Property	This stereotype could be applied to the property to indicate that it represents a Solidity mapping variable.
«member»	Property	This stereotype could be applied to the property to indicate that it represents a Solidity struct member.
«modifier»	Constraint	This stereotype could be applied to constraint to indicate that it represents a Solidity modifier.
«parameter»	Parameter	This stereotype could be applied to the operation parameter to indicate that it represents a Solidity function parameter.
«payable»	Parameter, Operation, Property	This stereotype could be applied to an operation, parameter or property to indicate that it represents a Solidity payable state mutability.
«struct»	Class	This stereotype could be applied to a class to indicate that it represents a Solidity struct.
«variable»	Property	This stereotype could be applied to a property to indicate that it represents a Solidity contract variable.
«view»	Property	This stereotype could be applied to the property to indicate that it represents a

		Solidity payable state mutability.

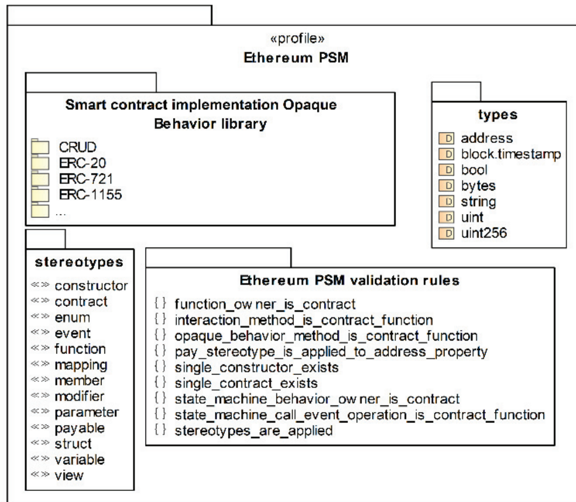


Figure 20. Ethereum PSM Profile

The overall composition of the structure defining stereotypes directly corresponds to the Solidity metamodel outlined in **Figure 28**. The Ethereum PSM profile types, stereotypes, validation rules and opaque behavior library were implemented in the MagicDraw CASE tool.

2.3.1.2. Hyperledger Fabric PSM profile

Similarly, as with the Ethereum PSM, the Hyperledger Fabric PSM is used to describe the structure and behaviour of a smart contract (it should be noted that, in the Hyperledger ecosystem, the smart contract is called the *chaincode*) intended to be hosted on the Hyperledger Fabric blockchain platform. The Hyperledger Fabric PSM is composed of the chaincode structure and the behaviour specification. Since the Hyperledger Fabric platform supports multiple programming languages as well, and the method's aim is to support the transformations to the executable code, the Hyperledger Fabric PSM is based on the Go programming language concepts. The metamodel is presented in **Figure 21** which represents how the UML metamodel elements are mapped to the Hyperledger Fabric Go chaincode stereotypes (**Table 13).**

Similarly, just as in the Ethereum PSM, the chaincode structure is specified by using the Class Diagram, and the chaincode behaviour can be specified by using the State Machine diagram. The State Machine diagram is used to describe the overall execution of the chaincode States, state Transitions, and the extension of the smart contract function behaviour. Additionally, at the Blockchain PSM level, the smart contract behaviour can be specified by using the Sequence diagrams (Interaction), and Opaque Behaviors. The Hyperledger Fabric PSM profile along with the UML metamodel is outlined in **Figure 19**, and the provided illustration mainly deals with

the specification of the chaincode structure. The UML metamodel for Interaction specification is elaborated upon in **Figure 19** as well. The Sequence diagram specifies an Interaction which consists of Lifelines, Messages, and Interaction Fragments, thus allowing the definition of a specific function behaviour, such as expressions, loops, conditional statements, and so on.

Figure 21. Hyperledger Fabric PSM. Extended UML metamodel

Like the Ethereum PSM profile, the Hyperledger Fabric PSM (**Figure 22**) also includes a number of datatypes specific to the Go programming language, and it also includes a library of Opaque Behaviors based on the standard implementations and the provided implementation code samples from documentation [119] [120]. This way, the Hyperledger Fabric PSM can be extended by selecting the provided Opaque Behaviors from the implementation library, or by specifying interactions (sequence diagrams).

Table 13. Hyperledger Fabric PSM stereotypes

Stereotype	UML Metaclass	Description
«argument»	Parameter	This stereotype could be applied to a class property to indicate that it represents a Go argument.
«chaincode»	Class	This stereotype could be applied to a class to indicate that it represents a Go chaincode.
«field»	Property	This stereotype could be applied to the property to indicate that it represents a

		Go structure field.
«function»	Operation	This stereotype could be applied to an operation to indicate that it represents a Go function.
«structure»	Class	This stereotype could be applied to a class to indicate that it represents a Go structure.
«variable»	Property	This stereotype could be applied to the property to indicate that it represents a Go variable.

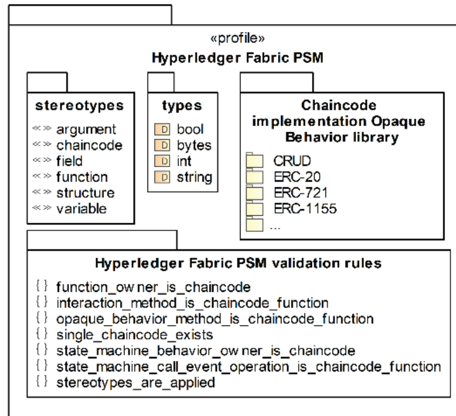


Figure 22. Hyperledger Fabric PSM Profile

Like the Ethereum PSM, the Hyperledger Fabric PSM stereotypes outlining the chaincode structural elements are based on the Go Chaincode metamodel provided in **Figure 29**. The Hyperledger Fabric PSM profile types, stereotypes, validation rules and the opaque behavior library were implemented in the MagicDraw CASE tool.

2.3.2. Blockchain PSM Definition

Like the Blockchain PIM specification, the Blockchain PSM definition encompasses three major steps. In the first step, the Blockchain PIM is transformed into the Blockchain PSM, and the transformation is automated by using model-to-model transformations implemented in ATL.

During the transformation, based on the specified smart contract structure and defined behaviour, at the Blockchain PSM level, additional smart contract elements are appended to the smart contract specification. The transformation algorithm for the Ethereum PSM and the Hyperledger Fabric PSM is detailed in the following sections. After the Blockchain PIM transformation, the transformed Blockchain PSM can be extended by the developer, and, lastly, the specified model validation takes place by using the OCL validation rules included in either the Ethereum PSM UML profile or in the Hyperledger Fabric PSM UML profile.

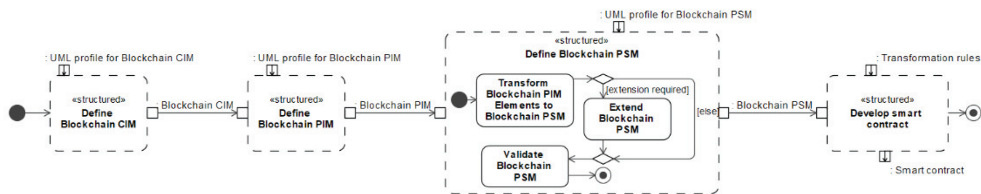


Figure 23. Detailed Blockchain PSM definition step in the context of the method

The blockchain PSM development steps are elaborated further in the upcoming sections.

2.3.2.1. Transformation of Blockchain PIM to Blockchain PSM

The smart contract structure from the blockchain PIM is transformed rather straightforwardly, mainly by applying platform-specific stereotypes. The main transformation deals with the extraction from the specified Blockchain PIM state machines.

Blockchain PIM to Ethereum PSM Transformation

As in the previously defined Blockchain PIM, the transformation to the Ethereum PSM is achieved similarly. The defined Blockchain PIM is used as an input for the transformation to the Ethereum PSM. Most of the transformation deals with the smart contract structure extension, during which, the smart contract is appended to include information which was specified in the State Machines. The complete transformation is presented in **Figure 24**, whereas the implementations are presented in *Appendix 2. Blockchain PIM to Ethereum PSM transformation rules*.

During the transformation, a PIM smart contract Class is transformed into a PSM contract, and any contained Classes are transformed into «struct» Classes and have «member» Properties with the Ethereum PSM types appended to them. Once the contained Classes transformation has been completed, Enumerations and its Literals are transformed, and they have an «enum» and «member» Property stereotype applied correspondingly. Similar actions are performed during the Class Properties and Operations transformations, and these become «variable» Properties and «function» Operations. Lastly, each State Machine is analysed, and, based on the information recorded, an «enum» Enumeration is created; additionally, based on the specified Transition Effects, new «event» Operations are created as well.

Additionally, during the transformation, the State Machine is modified, provided that the State Machine behaviour specifications exist. During the modification, the Ethereum Solidity-specific design patterns are applied to the smart contract if a smart contract has a denoted classifier Behavior State Machine. Regardless of the modification, the State Machine still outlines the relevant States, Transitions, Conditions, Call Event Operations and Effects. If it has been determined that the specific software patterns are applicable, during the transformation, the State Machine and the smart contract structure are modified to conform to the State Machine pattern, and/or to the timed transitions pattern. Additionally, any recurring

Transitions guard Constraints are removed, and a «modifier» Constraint is created, which is then applied to the relevant «function» Operations.

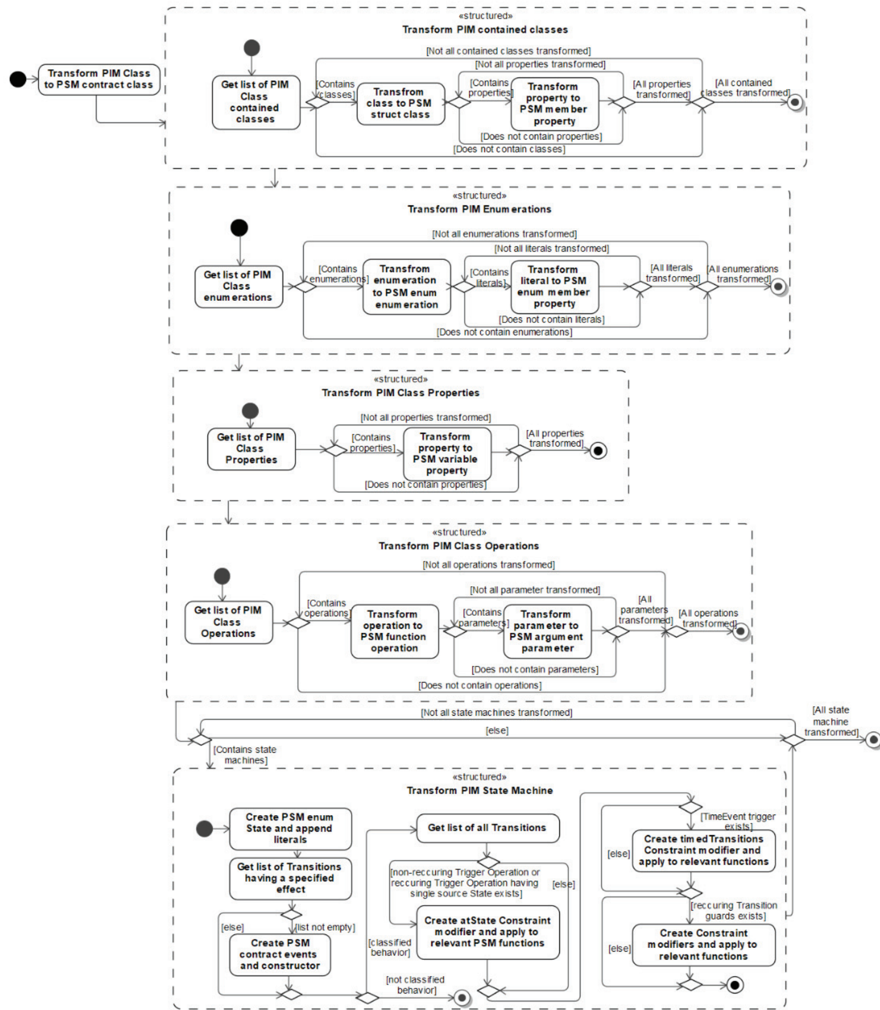


Figure 24. Blockchain PIM to Ethereum PSM Transformation

Based on the State Machine, if such exists, a new State Enumeration is created, which outlines all the possible States as Enumeration Literals. Then, the Transitions that have Effects specified are collected and analysed. During the analysis, if it is determined that an Effect is specified on the Transition whose source is the initial Pseudostate, and the constructor has not yet been created, it is transformed based on the specified Opaque Behavior Effect specification to the «contract» Class «constructor» operation. Any other Transition Effects are transformed into «event» Operations, again based on the specified Opaque Behavior Effect the «event» Operation can have additional Properties appended to it as well.

Next, all the Transitions are collected, and the Call Event usage is checked, and, if it is determined that the Operation was used no more than once per

Transition, or if used multiple times, but the source of the Transitions is still the same, the Solidity state machine design pattern is applied to the smart contract. In particular, an atState «modifier» Constraint is created, and dependency relationships between the «modifier», specific state «enum» Enumeration Literal (corresponds to the source of the transition), and the specific «function» Operation is created.

Additionally, if, in the Transition list, there exists a Time Event Trigger, a timedTransitions «modifier» is created. The «modifier» is responsible for determining if a specific point of time has been reached compared to the current timestamp. The Time Event Trigger cannot be translated directly into Solidity as it does not support time-based Transitions. The behaviour logic is recorded in the timedTransitions «modifier» Constraint body. For fully supporting the timedTransitions «modifier», an additional «variable» Property is created, which denotes the smart contract creation date. By using the creation date, any of the specific relative Time Event (after(x time)) can be exchanged with a derived value at(creationdate+ x time), then, this logic is recorded in the State Machine diagram, and the created timedTransitions «modifier» Constraint is applied to all smart contract «function» Operations.

Lastly, the state machine Transitions guards are analysed, and, if found, the recurring guards are removed from the state machine and applied to the relevant Transition Call Event Operations. The transformation result, the Ethereum PSM, then can be used by the developer, and it includes the contract specification as a Class diagram, provided that the State Machine behaviour was outlined, includes the State Machine behaviours, as well as additional structural elements («enum», «modifier», «constructor», «event») transformed based on the State Machine specification.

Blockchain PIM to Hyperledger Fabric PSM Transformation

As stated above, the defined Blockchain PIM is used as an input in the transformation to the Hyperledger Fabric PSM as well. Most of the transformation deals with the chaincode structure modification, provided that, during the Blockchain definition, the State Machines were specified. The full transformation is presented in **Figure 25**, and the implementations are presented in *Appendix 3. Blockchain PIM to Hyperledger Fabric PSM transformation rules*. Just like the Ethereum PSM, the Hyperledger Fabric PSM at the end of transformation consists of a UML Class diagram and State Machine specifications. The PSM Class diagram represents the «chaincode» structure, and, after transformation, compared to the Blockchain PIM, it is extended to include information from the State Machines. As well as being customized for the Go programming language, new types, denoted stereotypes from the provided Hyperledger Fabric PSM UML profile, were applied.

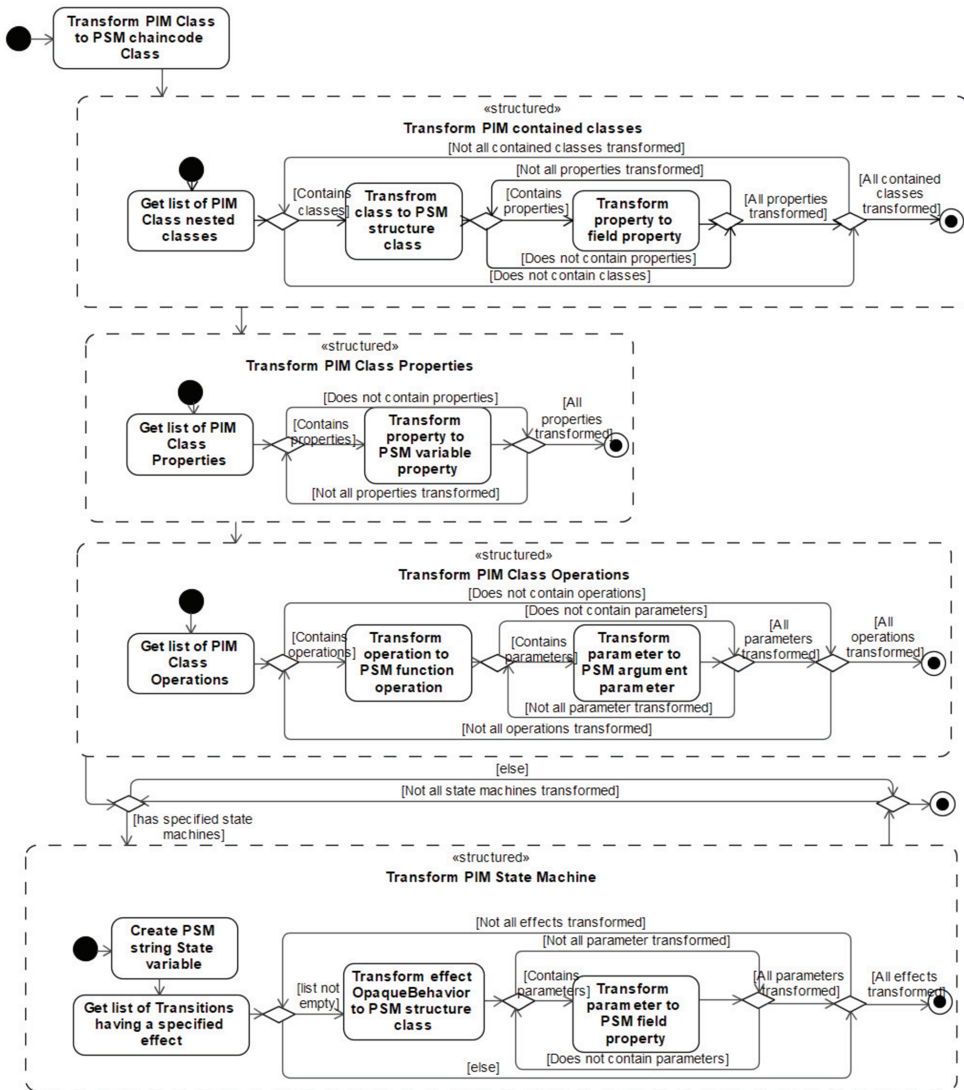


Figure 25. Blockchain PIM to Hyperledger Fabric PSM Transformation

At the start of the transformation, the PIM «contract» Class is transformed into the PSM «chaincode» Class, meaning that a new Class is created at the PSM level that has a «chaincode» stereotype applied to it. Similarly, as in the Ethereum PSM transformation, new «structure» Classes are created based on the contained Classes in the model, and each Property of the contained Class is transformed into the PSM «struct» Class Property with a «field» stereotype. Each structure Class also has «field» Properties appended to it, and the Property is tailored to the Go programming language by selecting a new data type. Once this has been done, the «chaincode» Class is appended with global «variable» Properties, which results from the direct transformation of the attributes of the Blockchain PIM smart contract

Class. Lastly, the Blockchain PIM Class Operations are transformed into Hyperledger Fabric PSM «function» Operations, with each «function» containing Operation Parameters if such existed in the Blockchain PIM.

Once the transformation of the structural elements has been completed, the specified State Machine Behaviors are analysed, if specified in the Blockchain PIM. Based on this, a string State «variable» Property is created which records the current state of the «chaincode» or the «structure» Class. This is done because the Go programming language does not support enumeration elements. Afterwards, the Transitions having a specified Effect are collected, based on which, new «structure» Classes are created. Any Effect that has a specified Opaque Behavior is transformed into a «structure» Class, and, if specified, the Opaque Behaviour Properties are transformed into the Class «field» Property.

The result of the transformation is the Hyperledger Fabric PSM, which consists of the Class and State Machine diagrams, and which can later be used by the developer for the transformation to the chaincode in the Go programming language.

2.3.2.2. Extending Blockchain PSM definition

After the transformation, a Blockchain PSM is provided to the developer. Once again, the developer may choose to augment the transformed Blockchain PSM. The smart contract structure can be extended by additional properties, operations, and data structures, regardless of whether it is tailored to the Ethereum PSM or the Hyperledger Fabric PSM. The smart contract can also be extended by using platform-specific elements like events, constructor, and modifiers.

Additionally, in order to support the current development practices during which developers rely on the standardized code implementation, the method allows to incorporate function implementations in the PSM definition step. After the transformation from the Blockchain PIM to the Blockchain PSM has been completed, at the Blockchain PSM level, the developer is provided with the capability to outline the function behaviour by specifying Interactions (Sequence diagrams).

The developer may select Opaque Behaviors from the provided curated smart contract implementation library to be included in the contract. The provided Opaque Behaviors have a specified body which includes the smart contract function code for a specific blockchain platform, and, once utilised, it can be extended by the behavioural logic specified in the State Machines. Each Opaque Behavior specified can also have Properties, Parameters, and additional Operations that need to be appended to the smart contract. This sort of extension is available to both the Ethereum, and the Hyperledger Fabric platforms, and it allows extending Blockchain PSMs and, ultimately, the smart contract code by reusing common relevant smart contract implementation code samples.

During or after the manual extension, the developer needs to validate the Blockchain PSM. The validation of the Blockchain PSM is used to ensure that the model does not have any correctness and completeness issues and that it can be successfully used for the model-to-text transformations, during which, the smart

contract code is produced. In particular, the model is checked for conformity to the expected Blockchain PSM structure. Both Blockchain PSMs profiles validation rules are similar in nature and are used to validate similar elements at the Blockchain PSM level.

Ethereum PSM Validation

During the validation of the Ethereum PSM model, the correctness and completeness necessary for the enablement of the model-to-text transformations are checked. The Ethereum PSM is checked for conformance to the expected model structure and is specialized to the Solidity programming language and its supported concepts. During validation, the PSM is checked to have a single «contract» Class, and a single «constructor» Operation specified. Any containing elements like «struct» Classes, «enum» Enumerations, Operations, State Machine behaviours, and Opaque Behaviors are checked to be contained by the «contract» Class. The stereotype usage is checked as well, during which, it is determined whether the stereotypes are applied, and, in some cases, whether they are applied correctly. Lastly, it is determined whether the specified Opaque Behavior, Interactions or State Machine Transition Call Events are correctly associated with the Blockchain PSM «contract» Class «function» Operations. Once validated, the Ethereum PSM is further used as an input for the transformation and development of the smart contract code in the Solidity programming language.

During the validation of the Ethereum PSM model, the correctness and completeness necessary for the enablement of the model-to-text transformations are checked. The full list of the Ethereum PSM validation rules can be seen in **Figure 26**, and some exemplary OCL implementations are presented in **Table 14**. While the provided validation rules list covers the relevant UML metamodel extensions and validates conformance to the expected model structure, such a list could be extended with utility validation rules to facilitate the model specification.

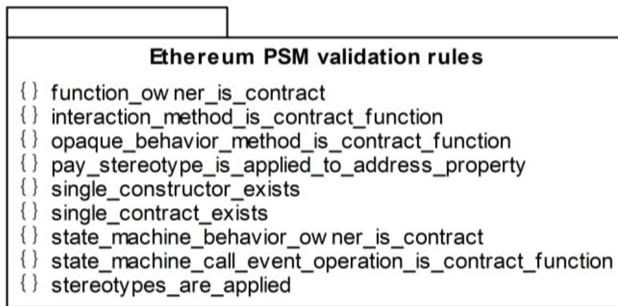


Figure 26. List of Ethereum PSM Validation Rules

The Ethereum PSM is checked for conformance to the expected model structure, during which, the PSM is checked to have a single «contract» Class, and a single «constructor» Operation specified. Any containing elements like «struct» Classes, «enum» Enumerations, Operations, State Machine behaviours, and Opaque Behaviors are checked to be contained by the «contract» Class. The stereotype usage

is checked as well, during which, it is determined whether the stereotypes are applied, and, in some cases, whether they are applied correctly. Particularly, when checking if the «pay» stereotype is applied correctly, only a single constructor exists.

Lastly, it is determined whether the specified Opaque Behavior, Interactions or State Machine Transition Call Events are associated with the Blockchain PSM «contract» Class «function» Operations. While the validation rules checking the structure are highly specialized to a specific platform, the validation rules related to the behaviour specification can be adapted to other platforms with minimal modifications.

Table 14. Implementation of Ethereum PSM Validation Rules

<table>
<tr><th>Validation Rule</th></tr>
<tr><td>context Operation inv function_owner_is_contract:
self.appliedStereotypeInstance.name='function' implies
(self.owner.appliedStereotypeInstance.name='contract' and self.owner.oclIsTypeOf(Class))</td></tr>
<tr><td>context Message inv interaction_method_is_contract_function:
self.messageSort=MessageSort::synchCall
implies (self.signature.oclIsKindOf(Operation) and
self.signature.appliedStereotypeInstance.name='function' and
self.signature.owner.oclIsKindOf(Class) and
self.signature.owner.appliedStereotypeInstance.name='contract')</td></tr>
<tr><td>context OpaqueBehavior inv opaque_behavior_method_is_contract_function:
self.specification.oclIsTypeOf(Operation) implies
(self.specification.appliedStereotypeInstance.name='function' and
self.specification.owner.appliedStereotypeInstance.name='contract')</td></tr>
<tr><td>context Property inv payable_stereotype_is_applied_to_address_property:
self.appliedStereotypeInstance.name='payable' implies self.type.name='address'</td></tr>
<tr><td>context Class inv single_constructor_exists:
self.appliedStereotypeInstance.name='contract' implies self.ownedOperation->collect(o |
o.appliedStereotypeInstance.name='constructor')->size() <= 1</td></tr>
<tr><td>context Model inv single_contract_exists:
self.ownedElement->collect(e | e.oclIsKindOf(Class) and
e.appliedStereotypeInstance.name='contract')->size()=1</td></tr>
<tr><td>context StateMachine inv state_machine_behavior_owner_is_contract:
self.owner.appliedStereotypeInstance.name='contract' and self.owner.oclIsTypeOf(Class)</td></tr>
<tr><td>context Trigger inv state_machine_call_event_operation_is_contract_function:
self.event.oclIsKindOf(CallEvent) implies
(self.event.oclAsType(CallEvent).operation.appliedStereotypeInstance.name='function' and
self.event.oclAsType(CallEvent).operation.owner.oclIsKindOf(Class) and
self.event.oclAsType(CallEvent).operation.owner.appliedStereotypeInstance.name='contract')</td></tr>
<tr><td>context Model inv stereotypes_are_applied:
self.ownedElement->forAll(e | (e.oclIsKindOf(Class) implies
(e.appliedStereotypeInstance.name='contract' or e.appliedStereotypeInstance.name='struct'))
or (e.oclIsKindOf(Property) implies (e.appliedStereotypeInstance.name='variable' or
e.appliedStereotypeInstance.name='member'))
or (e.oclIsKindOf(Enumeration) implies (e.appliedStereotypeInstance.name='enum'))
or (e.oclIsKindOf(Operation) implies (e.appliedStereotypeInstance.name='function' or
e.appliedStereotypeInstance.name='constructor' or e.appliedStereotypeInstance.name='event'))
or (e.oclIsKindOf(Constraint) implies (e.appliedStereotypeInstance.name='modifier'))
)</td></tr>
</table>

Once validated, the Ethereum PSM is further used as an input for the transformation and development of the smart contract code in the Solidity programming language.

Hyperledger Fabric PSM Validation

Similar validation rules as in the Ethereum PSM profile are provided in the Hyperledger Fabric PSM UML profile. The Hyperledger Fabric PSM is checked for conformance to the expected model structure and is tailored to the Go Chaincode supported concepts. The model is checked for completeness and correctness to support the model-to-text transformations. Hyperledger Fabric PSM is validated for the stereotype usage, and any necessary Elements that do not have stereotypes denoted are marked for the developer. Additionally, the Hyperledger Fabric PSM is checked for correspondence to the expected model structure. Model Elements, like Operations, «structure» Classes, Opaque Behaviors, and Interactions are to be contained by the «chaincode» Class. Furthermore, the Behaviour specifications: Opaque Behaviour, Interaction, and Call Event usage, are checked for association to the specific «chaincode» Class «function» Operations. The validated Hyperledger Fabric PSM is used as an input for the chaincode development in the Go programming language, which starts with the model-to-text transformation.

With similar validation rules as in the Ethereum PSM profile, the Hyperledger Fabric Examples are provided alongside the Hyperledger Fabric PSM Profile. The model is checked for completeness and correctness in order to support the model-to-text transformations. The list validation rules can be viewed in **Figure 26**, and some exemplary OCL implementations are presented in **Table 15**. Although the list of the provided validation rules covers the necessary UML metamodel extensions and ensures that the model structure adheres to the expected structure, additional utility validation rules could still be included to support the process of model specification.

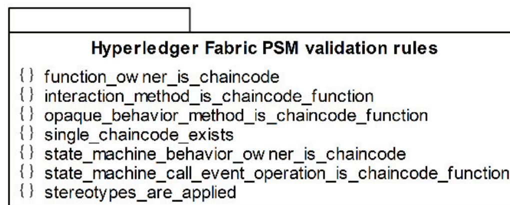


Figure 27. List of Hyperledger Fabric PSM validation rules

The Hyperledger Fabric PSM is validated for stereotype usage, and any necessary Elements that do not have stereotypes denoted are marked for the developer. Additionally, the Hyperledger Fabric PSM is checked for correspondence to the expected model structure. Model elements, like Operations, «structure» Classes, Opaque Behaviors, and Interactions are to be contained by the «chaincode» Class. Furthermore, the Behaviour specifications: Opaque Behaviour, Interaction, and Call Event usage, are checked for association to the specific «chaincode» Class «function» Operations. The outlined validation rules related to the behaviour

specification are quite similar in nature to the Ethereum PSM rules and can easily be adapted to other platforms.

Table 15. Implementation of Hyperledger Fabric PSM Validation Rules

Validation rule
context Operation inv function_owner_is_chaincode: self.appliedStereotypeInstance.name='function' implies (self.owner.appliedStereotypeInstance.name='chaincode' and self.owner.oclIsTypeOf(Class))
context Message inv interaction_method_is_chaincode_function: self.messageSort=MessageSort::synchCall implies (self.signature.oclIsKindOf(Operation) and self.signature.appliedStereotypeInstance.name='function' and self.signature.owner.oclIsKindOf(Class) and self.signature.owner.appliedStereotypeInstance.name='chaincode')
context OpaqueBehavior inv opaque_behavior_method_is_chaincode_function: self.specification.oclIsTypeOf(Operation) implies (self.specification.appliedStereotypeInstance.name='function' and self.specification.owner.appliedStereotypeInstance.name='chaincode')
context Model inv single_chaincode_exists: self.ownedElement->collect(e | e.oclIsKindOf(Class) and e.appliedStereotypeInstance.name='chaincode')->size()=1
context StateMachine inv state_machine_behavior_owner_is_chaincode: self.owner.appliedStereotypeInstance.name='chaincode' and self.owner.oclIsTypeOf(Class)
context Trigger inv state_machine_call_event_operation_is_chaincode_function: self.event.oclIsKindOf(CallEvent) implies (self.event.oclAsType(CallEvent).operation.appliedStereotypeInstance.name='function' and self.event.oclAsType(CallEvent).operation.owner.oclIsKindOf(Class) and self.event.oclAsType(CallEvent).operation.owner.appliedStereotypeInstance.name='chaincode')
context Model inv stereotypes_are_applied: self.ownedElement->forAll(e | (e.oclIsKindOf(Class) implies (e.appliedStereotypeInstance.name='chaincode' or e.appliedStereotypeInstance.name='structure')) or (e.oclIsKindOf(Property) implies e.appliedStereotypeInstance.name='variable') or (e.oclIsKindOf(Operation) implies e.appliedStereotypeInstance.name='function'))

Once validated, the Hyperledger Fabric PSM is used as an input for the chaincode development in the Go programming language, which starts with the model-to-text transformation.

2.4. Transformation of Blockchain PSM to Smart Contract Code

2.4.1. Smart Contract Metamodel

Two smart contract metamodels for different platforms were outlined in order to determine the concepts that would need to be mapped in order to support code-generation activities. The method supports not only the specification of the smart contract artefact structure, but the behaviour specification elements as well.

2.4.1.1. Ethereum Solidity Metamodel

Ethereum supports multiple programming languages; and Solidity, while being domain-specific, also remains the most popular one. Additionally, the proposed smart contract standards are also based on the Solidity programming language [30].

As a result, to support the model-to-text transformations, a Solidity code metamodel was specified to outline the main concepts of the Solidity programming language (**Figure 28**). The main elements that contribute to the smart contract structure specification are Contract, Variable, Struct, Modifier, Enum, Function, Constructor, and Event. The behaviour is mainly expressed by using the Block, statements like Emit, Function Call Expressions, and so on. This behaviour specification is mainly used to outline the function and Constructor behaviour.

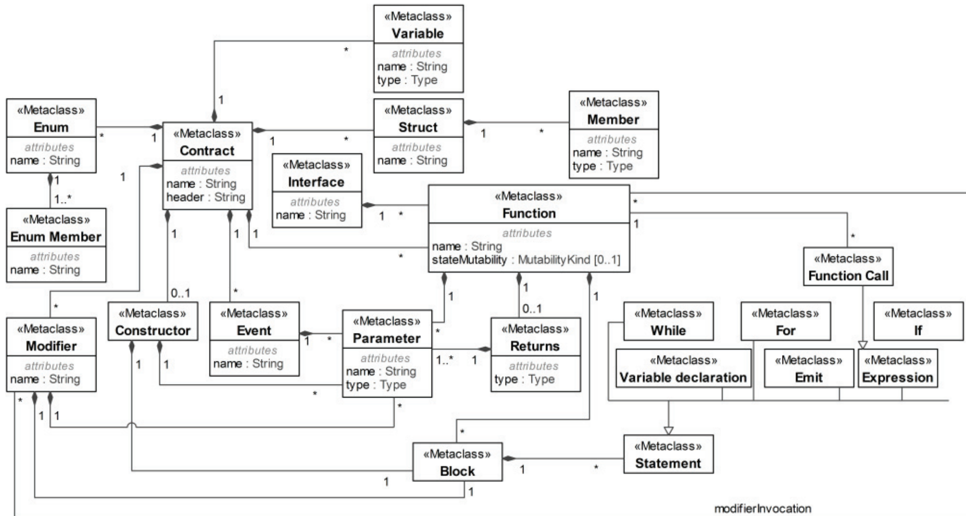


Figure 28. Solidity smart contract metamodel

The developed metamodel was used during the implementation for the M2T transformations from the Ethereum PSM to the Solidity programming language smart contract code.

2.4.1.2. Hyperledger Fabric Go metamodel

The Hyperledger Fabric platform supports Java, JavaScript and Go programming languages. Since the Hyperledger Fabric platform is also built by using the Go programming language, which makes the Go programming language the most supported one, Go was selected as the target language for transformations. The Go programming language is an object-oriented programming language, therefore, for analysis purposes, a metamodel for Go, specifically, Chaincode, was outlined (**Figure 29**). The metamodel mainly outlines the structural elements of the Go programming language, such as Chaincode, Function, Structure, Argument, Variable, and so on. And, since the method supports behaviour specification, the behavioural elements that can be generated by using model-to-text transformations are mainly described by the expression and its subclasses, such as Function Calls, and Statement.

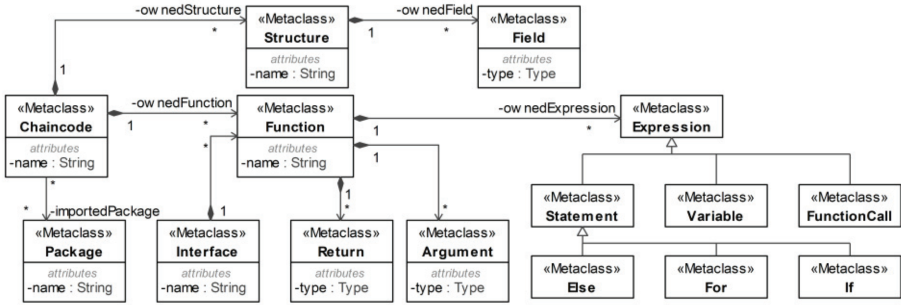


Figure 29. Go chaincode metamodel

The developed metamodel was used during the implementation for the M2T transformations from the Hyperledger Fabric PSM to the Go programming language smart contract code.

2.4.2. Smart Contract Development

The last step in the MDA-based method is the development of the smart contract which starts with the model-to-text transformation which produces a smart contract code. As previously stated, the smart code is produced from the previously developed Blockchain PSM, just this time by using model-to-text transformations. In both the Ethereum and Hyperledger Fabric cases, the generated smart contract code can be extended by the developer manually or deployed to the specific blockchain platform (**Figure 30**).

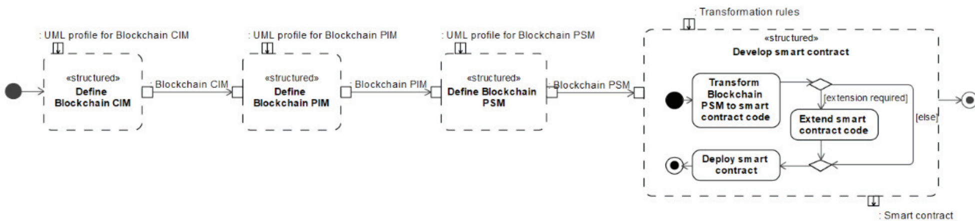


Figure 30. Detailed Blockchain PIM definition step in the context of the method

The presented model-to-text transformations are implemented by using the Eclipse Acceleo plugin. Acceleo is a MOFM2T language implementation, which, together with the Acceleo plugin, allows developing the MOFM2T template for transformations. The templates are supplied with a Blockchain PSM XMI file which is interpreted to produce either the Hyperledger Fabric chaincode code in the Go programming language, or the Ethereum smart contract in the Solidity programming language. The transformation implementation is presented in *Appendix 4. Ethereum PSM to Solidity smart contract transformation templates* and *Appendix 5. Hyperledger Fabric PSM to Go chaincode transformation templates*.

2.4.2.1. Blockchain PSM to smart contract code transformation

Transformation of Ethereum PSM to Solidity smart contract code

The transformation from the Ethereum PSM to the Solidity smart contract (**Figure 31**) starts with the creation of a smart contract file, which is then appended line by line. The transformation continues with the Class structure production, during which, the smart contract class attributes, structures, and their attributes are appended, provided these were specified for the smart contract. Additionally, any outlined Enumerations are generated, and Constraints are transformed into modifiers.

Once most of the smart contract structure has been generated, the transformation of the smart contract behaviour begins. During the behaviour generation, the Operations which do not need to be extended in using the information specified in the State Machines are appended to the smart contract. Particularly, the constructor, function, and events are generated. The generation starts with the Operation header based on the stereotype, either one of the three is generated, which outlines the Operation name, the Parameters and the applied modifiers. Then, the Operation is appended with a specified behaviour, which is generated either from a specified Opaque Behavior specification if any behaviours were included in the smart contract from the standard library, or generated from the specified Interaction if a function behaviour was specified by using a Sequence diagram.

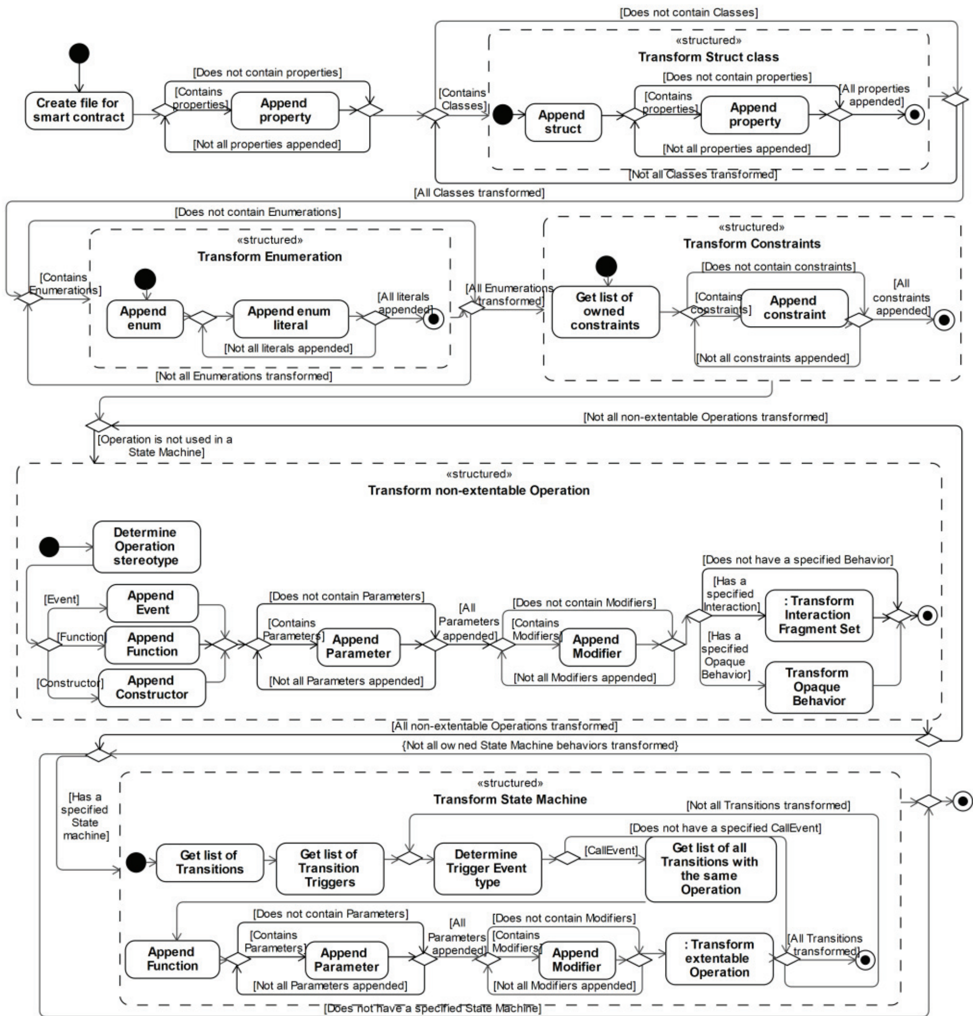


Figure 31. Transformation of Ethereum PSM to Solidity smart contract code

The Interaction transformation to the Solidity function body starts with the analysis of Interactions fragments (**Figure 32**). The Combined Fragment is transformed into a (for, while, if) statement. Meanwhile, the Message Occurrence specification is transformed into a variable declaration, function call expression, or emit statement other expressions. Additionally, after any transformation that does not append the Message Occurrence specification, a transformation is called recursively to ensure that the order of the statements is retained.

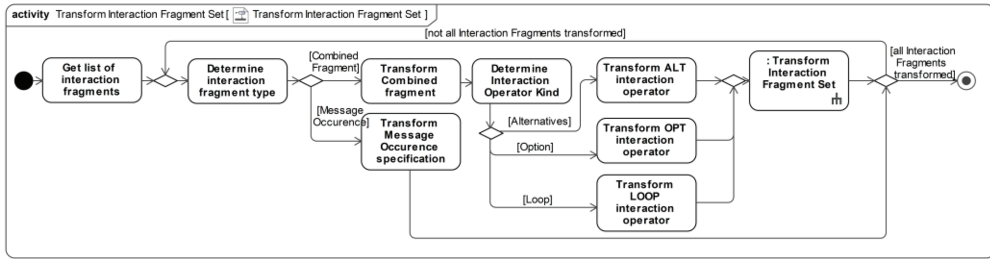


Figure 32. Transformation of the InteractionFragment set into the Solidity Function code

After non-extendable Operations have been generated, the State Machines are analysed, and the functions that were specified as Transition Trigger Call Events are extended by the behaviour specified in the State Machine. During this step, the State Machine behaviour is analysed, then, each State Machine Transition with its Event Triggers is collected, and each Call Event that shares the same Operation is used for the specific Operation code generation. Like previously, concerning the function header, Parameters and modifiers are appended, and then the smart contract function behaviour generation starts.

Each Operation transformation mainly consists of the transformation of the Transitions sources and Transitions targets (**Figure 33**). Additionally, if an Operation has a specified Opaque Behavior, the behaviour is appended twice, in parts, depending on the Opaque Behavior body specification a return statement position.

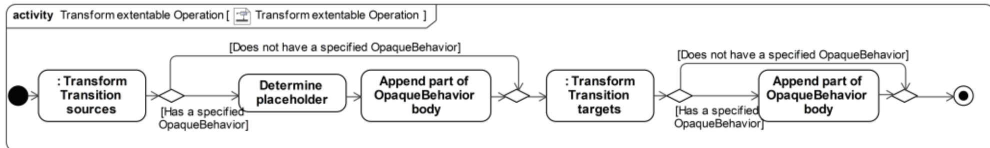


Figure 33. Transformation of Transition Operation into the Solidity Function code

The Transition sources are transformed by using all the Transitions sources, and by appending conditional statements based on the source (**Figure 34**). The conditional statement is appended while checking whether a specific state is achieved in order to perform a specific Transition/function behaviour, and this conditional statement is *not* appended if the Transition is from the initial Pseudostate kind. Additionally, the conditional expression is extended with an operator and the guard Constraint if such a Constraint exists. Similarly, the specific function behaviour is appended by conditional statements if the Transition source is a junction Pseudostate. Before that, though, the source of each Transition connected to the junction Pseudostate needs to be determined as well in order to append the conditional statements and the Transition guard Constraints – if such exist.

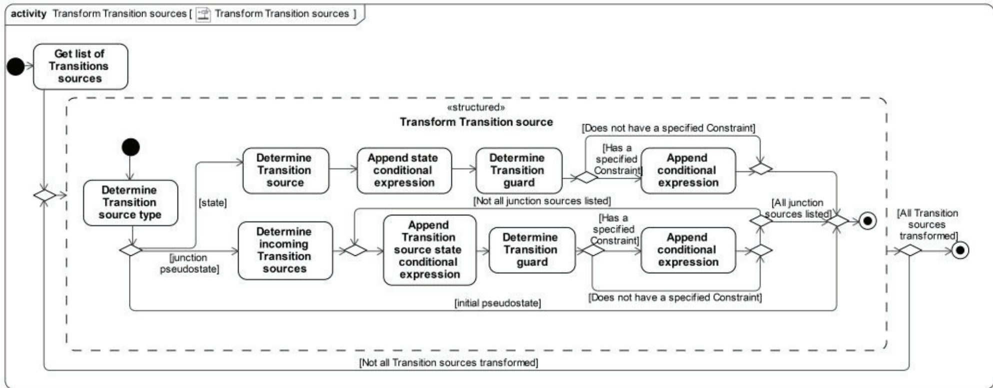


Figure 34. Transformation of Transition Sources into the Solidity Function code

After the Transition sources have been appended, the function behaviour is appended by the Opaque Behaviour body (**Figure 33**). And, in the same manner, as the Transition sources, the Transition targets are appended (**Figure 35**). This time, the Transition targets are transformed *not* into conditional statements, but into state declaration expressions. Depending on how many Transitions there exist with the shared Call Event Operation, the transformation may start with a conditional statement if more than two Transitions share the same Call Event Operation. Then, the Transition source is determined, and the conditional expression is appended with a guard Constraint provided that it was specified. Otherwise, the conditional expression is skipped, and the target transformation starts.

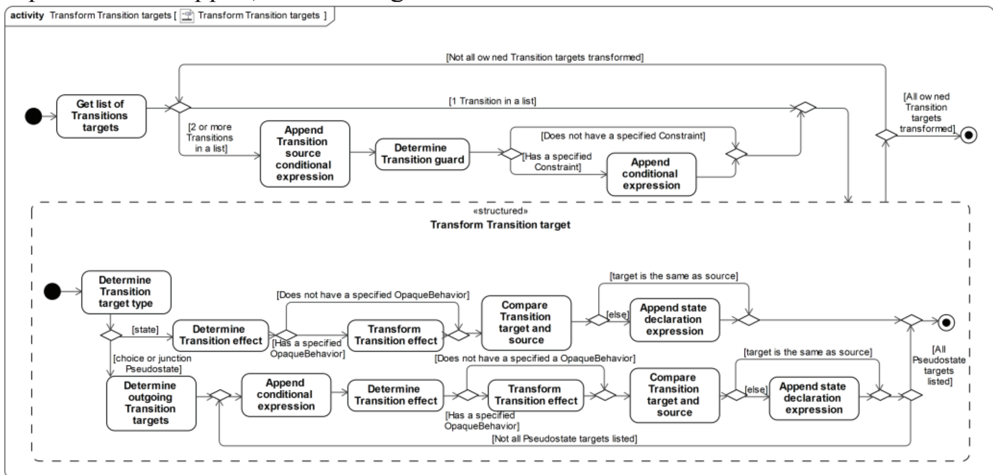


Figure 35. Transformation of Transition targets set into the Solidity Function code

Similarly as with Transition sources, the target may be a State; then, if the Effect has been specified, the Effect expression is appended, and the state declaration is appended if the source and the target do not match. Alternatively, if the Transition target is a choice or a junction Pseudostate, additional outgoing

Transition targets are determined and conditional expressions, alongside the Effect – if specified – and the state declaration expression are appended.

Transformation of Hyperledger Fabric PSM to GO

Similarly, as in the generation of the Ethereum Solidity smart contract codes, the Hyperledger Fabric PSM transformation to the Go chaincode starts with the creation of a chaincode file (**Figure 36**). The file is appended line by line by analysing the defined Hyperledger Fabric PSM. Since the Go programming language supports fewer concepts compared to the Solidity metamodel, the chaincode code generation in terms of the structure mainly consists of the Property and the struct Class generation.

Once the structure has been appended, the non-extendable Operations are transformed. So, either a function with the Parameters, basically the header of a function, is appended, or a function with the function body is appended. The behaviour can be specified in two ways, an Opaque Behavior body code embedding is appended, or an Interaction is analysed and transformed into the software code (**Figure 37**).

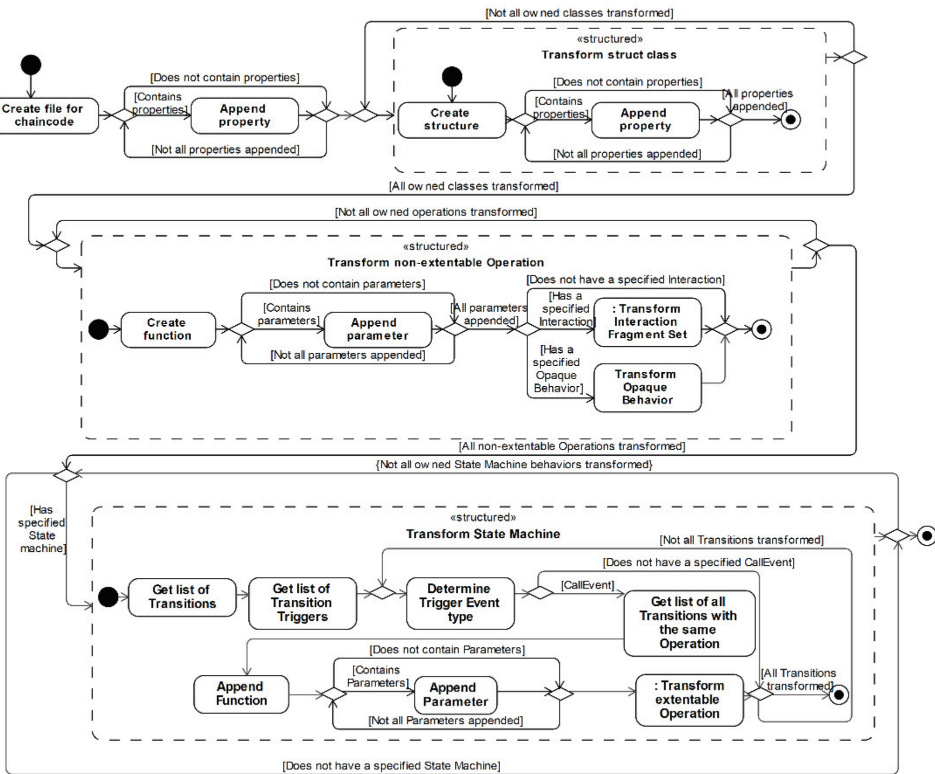


Figure 36. Transformation of Hyperledger Fabric PSM to Go chaincode

Particularly, during the Interaction analysis, the function's body is appended line by line, and, based on the Interaction Fragment type, either a conditional (if, else), or loop (for) statement is appended, or, based on the Message Occurrence

specification, a variable declaration, function call, or other expressions are appended to the function source code. Additionally, the same transformation step is called recursively to ensure that the hierarchy of statements is maintained as specified in the Sequence diagram.

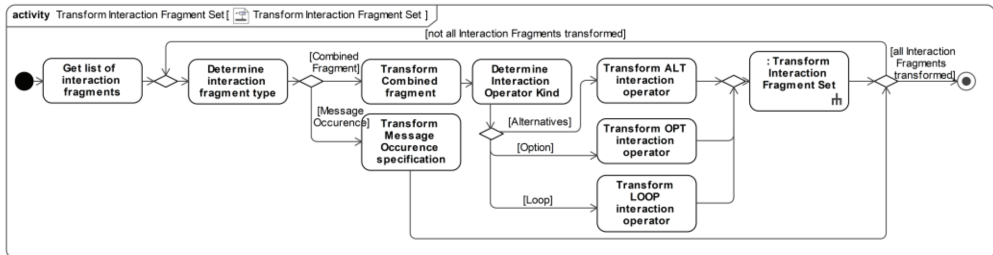


Figure 37. Transformation of InteractionFragment set into the Go function code

Once the non-extendable Operations transformation has been done, the State Machine Behavior is analysed. The execution of this is similar to the behaviour specified in the previous section. The State Machines are analysed by collecting the Transition Triggers, and, if it is determined that Operation Call Events do exist, the Event that shares the same Call Event Operation are used to define the behaviour of an extendable function. The function header is appended, with the function name, and the Parameters are appended to the file, and, based on the usage in the State Machine, the behaviour is appended to the chaincode file (**Figure 38**).

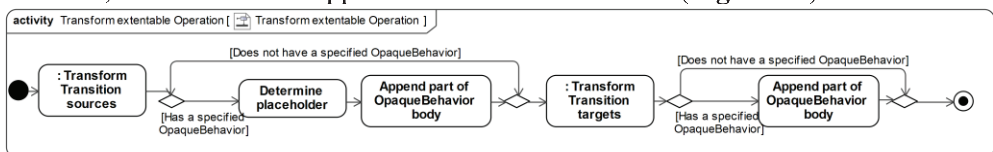


Figure 38. Transformation of state machine behaviour into the Go function code

As stated above, the function behaviour is appended with conditional statements which check whether the chaincode is in the correct state to trigger a specific transition (**Figure 39**).

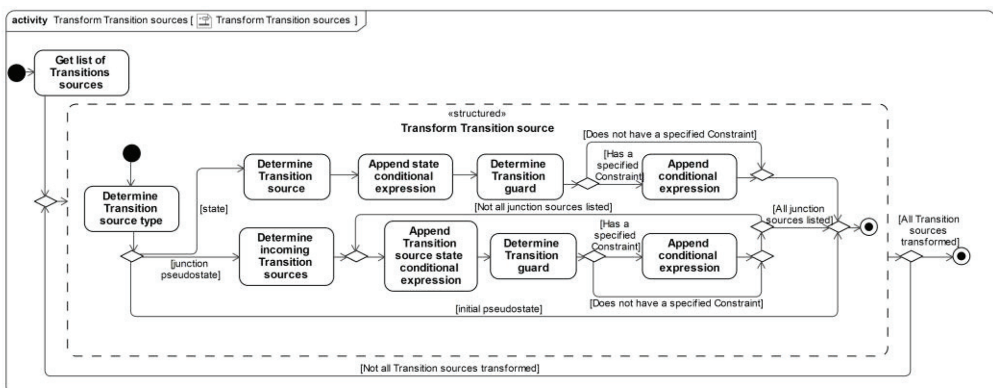


Figure 39. Transformation of Transition sources set into the Go Function code

Once the conditional statements based on the transition sources have been appended, the part of the Opaque Behavior body is appended, if specified. Subsequently, the state declarations based on the transition sources are generated (**Figure 40**). Specifically, the go chaincode file is an appended state declaration statement which records the Transition to another State, provided that the specified transition has a different source and target.

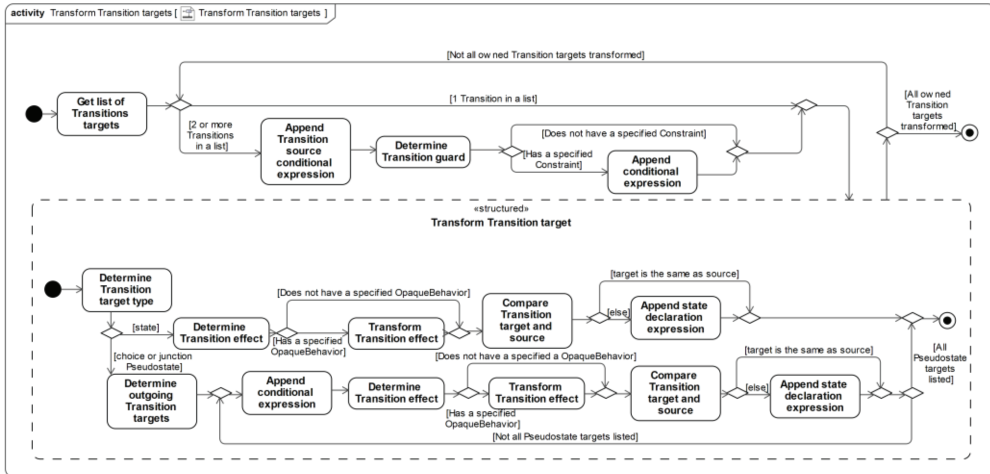


Figure 40. Transformation of Transition target set into the Go Function code

The proposed MDA-based method for smart contract-based system development transformations, including model-to-model transformations and model-to-text transformations, is evaluated in the following chapter.

The main differences between the platforms lie in an overview of the supported smart contract elements. Considering that Solidity is a more comprehensive language specifically tailored for smart contract development, the method and the model transformation deal with a wider element number which needs to be supported. While the chaincode development is mostly dependent on the specific language, and, as defined previously, the Go programming language is not as comprehensive, and it offers a narrower support of the smart contract-specific concepts.

2.5. Method extension

The method has been developed for two platforms, namely, Ethereum and Hyperledger Fabric (**Figure 41**), but it can be tailored to other platforms as well. The activities following the Define Blockchain PIM should be supported by additional method extensions.

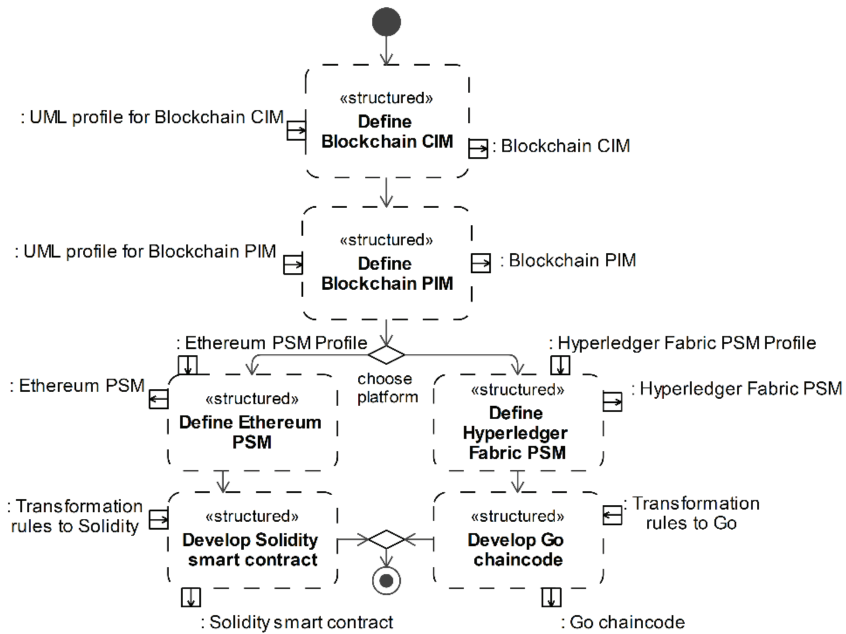


Figure 41. MDA-based method implementation

The method extension steps include such activities as:

1. Identify the specific concepts and requirements of the Blockchain PSM, determining how MDA can be used to model and automate the development of the specific technology smart contracts.
2. Define the Blockchain PSM specification extension profiles; additionally, the developed profile needs to be mapped to the UML metamodel.
3. Define the Blockchain PSM technology metamodel; the specific concepts of selected technology should be defined in the metamodel to be used in model transformations to the implementation code.
4. Define the Transformations from the Blockchain PIM to Blockchain PSM, which, based on the already provided Blockchain PIM structure, would be used to produce the newly defined Blockchain PSM.
5. Define the Transformations from the Blockchain PSM into a code, this is the final step in which the executable code is produced; this transformation is closely related to the comprehensiveness of the implementation language

The aforementioned steps of the MDA method extension in terms of the Blockchain PSM metamodel and the transformation definition have been performed for both the Ethereum and the Hyperledger Fabric platforms for the development of the Ethereum PSM and the Hyperledger Fabric PSM.

2.6. Limitations of the method

Although the introduction of the modelling approach into the smart contract-based system development can automate the design and implementation activities, and this introduces additional complexity as a considerable amount of time should be spent modelling the structure and the behaviour of a smart contract. The proposed MDA method requires an additional effort for refining the structure and behaviour at each abstraction level since a more comprehensive model will be a basis for a more comprehensive generated implementation smart contract code. Unfortunately, the model developer is limited in terms of the design expressiveness compared to other model-based methods, as the model needs to adhere to the strict rules to produce an executable smart contract code. Additionally, considering that there is not one single 'correct' way to use UML, developers may be accustomed to using UML in different ways from those outlined by the method.

At the CIM level, the developer at minimum is required to outline business processes. By using the outlined business processes, the developer must specify the use case model, as any subsequent model transformations are dependent on it, and they can additionally specify the domain model. The use case and domain model definition, based on business processes, is not automated, as a number of design decisions need to be made by the developer.

After the transformation to the Blockchain PIM, the model should be refined and manually extended by specifying the behaviour of the smart contract using state machines based on CIM business processes. Again, transformation from business processes needs to be performed manually to ensure that the intended behaviour is relocated to the smart contract behaviour correctly.

At the generated Blockchain PSM level, the function behaviour could be outlined as well. At this level, sequence diagrams can be employed for specifying a function body, but the sequence diagrams can become very complicated and difficult to model. Furthermore, if the developer employs only the automated transformations and does not outline the behaviour of the smart contract (at the PIM or PSM level), the generated code will only include structural elements.

The method is currently only tailored for Ethereum and Hyperledger Fabric platforms, yet supports the possibility to introduce new platforms by following the rules defined in the previous section, although it is worth noting that such an extension requires considerable investment by the developer. Finally, the scope of the work deals mainly with the smart contract code production from the specified model, while the other smart contract-based system development activities remain unsupported. In the future, other development phases could be considered as well, e.g. testing and deployment.

III. EXPERIMENTAL EVALUATION

The experimental assessments have been conducted in two parts for evaluating the proposed MDA-based method implementation. In the first part, an evaluation of the smart contract code artefacts generated by the proposed model transformations is carried out. The second part focuses on the full method application and evaluation of the generated artefacts in the overall development process of the smart contract-based system.

3.1. Generated smart contract code evaluation

The main purpose of the evaluation is to demonstrate how the employment of the implemented MDA-based method for blockchain-based system development transformations can be used to generate the Solidity smart contract and the Go chaincode source code from the specified models. For this purpose, one section has been outlined for evaluating Solidity smart contracts for the Ethereum platform (the experiment results were also published in a paper [121]) whereas the other is for the Go chaincode for the Hyperledger Fabric platform. In both cases, smart contracts were modelled based on the smart contract implementations provided in the Solidity [118] and the Hyperledger Fabric [119] documentations.

The smart contracts were modelled by using the MagicDraw CASE tool employing the Blockchain PIM UML profiles. The specified models were exported as an XMI file, and, in both cases, the model-to-model transformations were performed by using the Eclipse ATL tool. Transformations were used to transform the defined Blockchain PIM to either the Ethereum PSM or the Hyperledger Fabric PSM.

The transformed Ethereum PSM and the Hyperledger Fabric PSM were imported into the MagicDraw CASE tool and extended by Opaque Behaviors from the profile, or were extended by specifying Sequence diagrams. Then both of the Blockchain PSMs were once again exported as XMI files and used as input for the model-to-text transformations implemented in the Eclipse Acceleo tool by using the MOFM2T templating language. The M2T transformations generated the Solidity smart contract and the Go programming language chaincode code whose evaluation in terms of code metrics and execution is presented at the end of each section.

3.1.1. Ethereum Solidity Smart Contracts

Three smart contract implementations (SimpleAuction, Purchase, and StateMachine) from Solidity documentation were selected which employ the state machine-like behaviour [118]. Following the Blockchain PIM definition rules, the smart contract structure using the class diagram and the smart contract behaviour using State Machine diagrams were modelled. For each Blockchain PIM, model-to-model transformations were performed, during which, the structure of the smart contract was extended at the Ethereum PSM abstraction level. Following the Blockchain PIM transformation to the Ethereum PSM, the SimpleAuction, Purchase, and StateMachine smart contract Solidity code was generated from each Ethereum PSM smart contract specification.

Lastly, for each generated Solidity smart contract, Solidity code metrics were calculated by using the VSCode tool, Solidity Metrics Plugin [122]. Each generated smart contract code was compared with the original in terms of the source code lines, complexity scores, the abstract syntax tree (AST) element count, and, in addition, similarity scores were calculated by using the SmartEmbed tool [41]. Finally, the generated and original smart contracts were compiled by using Remix [123], automatically generated Unit tests were run, and then deployed to JavaScript VM; finally, the execution costs of the smart contract were calculated.

3.1.1.1. SimpleAuction Smart Contract

The Simple Auction smart contract is provided in the Solidity documentation [124]. While the original smart contract does not have a specified state enumerator, it has a bool variable ended for tracking the smart contract status. By using this information, a smart contract structure (**Figure 42**, left) and behaviour using the State Machine were specified by using the Blockchain PIM definition rules. The variable ended in the context of this smart contract specification was replaced by two Open and Ended States in the State Machine diagram (**Figure 42**, right), and the Transitions were specified as Call Event Operations which change the status in the original smart contract. In total, the State Machine diagram includes two Call Event Operations, two guard Constraints, and three Effects.

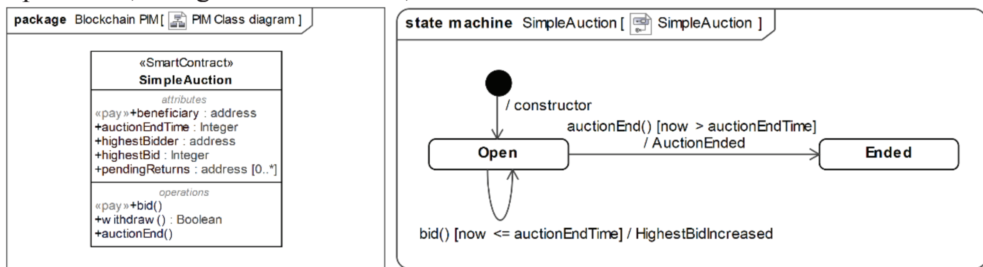


Figure 42. Blockchain PIM SimpleAuction smart contract

The specified Blockchain PIM SimpleAuction was transformed into the Ethereum PSM. During the transformation, the SimpleAuction class was extended by an additional state variable, and pendingReturns was transformed into a mapping variable, which includes a key and a value, and an Enumeration, with two Literals based on the two states in the PIM State Machine diagram. All the attributes, properties, and arguments had their types changed to Solidity-specific ones. Additionally, a constructor and two event Operations were created based on the specified Effect Opaque Behaviors in the PIM State Machine diagram, and, lastly, an atState modifier was generated and applied to functions whose execution depends on a specific state. The transition guards specified in the State Machine diagram were updated with block.timestamp instead of the now expression.

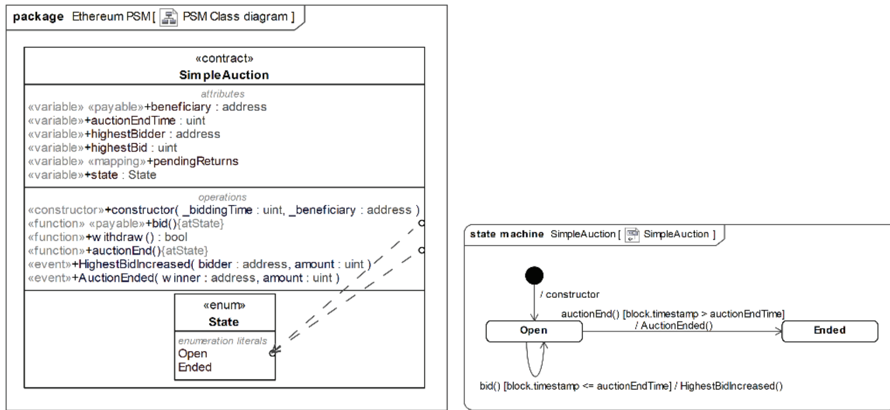


Figure 43. Ethereum PSM SimpleAuction smart contract

Ultimately, by using implemented model-to-text transformations, the specified Ethereum PSM was transformed into the Solidity smart contract code. Both the generated and the original SimpleAuction smart contracts were tested by using automatically generated unit tests which returned the same testing results.

A more thorough comparison of the generated and original SimpleAuction smart contract code is included in **Table 16**. The generated version from the original differs in terms of the AST element count. The generated smart contract includes more block, enum definition, enum value, variable declaration, and expression statements. The change is manifested because of the behavioural logic from the state machine diagram, specifically, the state enum and atState modifier invocations. The similarity score calculated by using the SmartEmbed tool shows that the smart contracts are similar as they reached a similarity score of 90% and a cosine similarity of 99%.

Table 16. Comparison of SimpleAuction Smart Contract Code

Source Units	**Generated**	**Original**
nSLOC	51	52
Complexity Score	30	28
AST Element	**Generated**	**Original**
Block	8	7
EmitStatement	2	2
EnumDefinition	1	0
EnumValue	2	0
EventDefinition	2	2
ExpressionStatement	15	13
FunctionCall	9	9
FunctionDefinition	4	4
Mapping	1	1
ModifierDefinition	1	0
ModifierInvocation	2	0

StateVariableDeclaration	6	6
VariableDeclaration	15	14
Similarity		
SmartEmbed Similarity Score	0.897	
Cosine Similarity	0.989	
Smart Contract Function	**Generated**	**Original**
constructor+deployment	606556	641548
bid()	71547	69331
withdraw()	23703	23636
auctionEnd()	60731	60651

The generated and the original SimpleAuction smart contract code was deployed on the JavaScript VM to calculate the smart contract execution costs. The Ethereum platform for execution of smart contract function attributes gas costs, and these were calculated for the deployment of the smart contract, which includes the constructor function call as well as other relevant functions. The estimated gas execution costs compared to the original are reduced by 5% for the deployment and increased for the functions at the most by 3% for the bid() function.

3.1.1.2. Purchase Smart Contract Code Generation

The Purchase smart contract [125] originally outlines a state machine-like behaviour where an asset goes through several states, and the functions are used as Transitions. So, like previously, the Purchase smart contract was specified by using the Class diagram (**Figure 44**, left) and the Blockchain PIM profile, while outlining 3 Properties and 4 Operations, and the elements dealing with the cryptocurrency transfer were specified by the «pay» stereotype. The main specification of the Purchase smart contract lies in the State Machine behaviour (**Figure 44**, right). The smart contract includes four States (Created, Locked, Inactive, and Release) and outlines four different transitions between them. Each Transition besides the initial to the first state has a specified Call Event Operation, guard Constraint, and Effect.

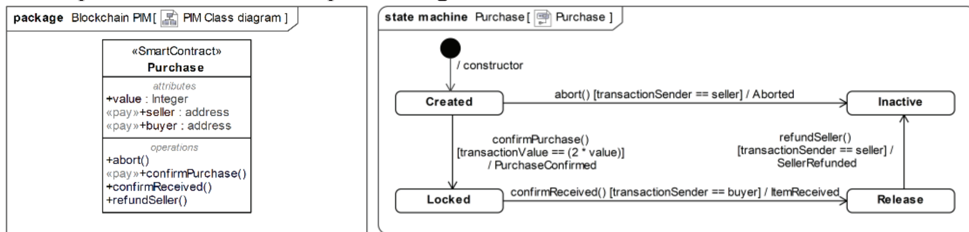


Figure 44. Blockchain PIM Purchase smart contract

The results of the transformation from PIM to PSM from blockchain to Ethereum are provided in **Figure 45**. During the transformation, the four states were transformed into an Enumeration and four Literals and a state variable, two atState and onlySeller additional modifiers were created and applied to all functions. The Purchase smart contract also demonstrates how, during the transformation, Solidity modifiers are produced based on specific recurring guard specification applications

to multiple call events. The onlySeller modifier is applied to two functions based on the usage of Transitions that the Operations were specified as Call Events; still, it is worth noting that, by default, the modifier is named mod1, as the naming was changed for comprehensibility purpoes. Additionally, based on the Effects, a constructor and four Aborted, PurchaseConfirmed, ItemReceived, and SellerRefunded events were created. The State Machine diagram had recurring guards removed, and the transactionSender and transactionValue were replaced by Solidity-specific msg.sender and msg.value.

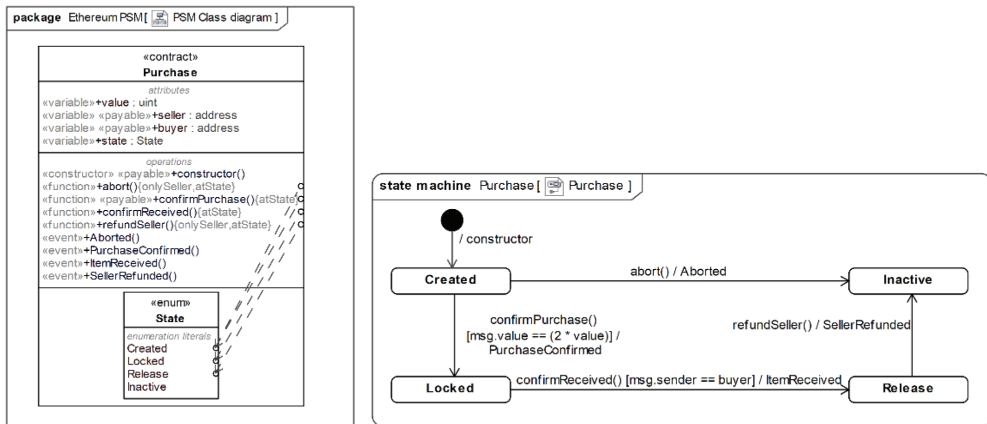


Figure 45. Ethereum PSM Purchase smart contract

As noted previously, the pair of Purchase smart contracts were compared. The generated Purchase smart contract code contains fewer code lines and obtains a lower complexity score calculated by VSCodetool. The AST element count also varies between the original and the generated smart contract code, specifically block, variable declaration, modifier, modifier invocation, and an expression statement. This is due to the fact that the generated smart contract contains one less modifier onlyBuyer based on a guard constraint and one less generic condition modifier for the generic conditional statement. Otherwise, the estimated similarity calculated by the SmartEmbed tool and the cosine similarity reaches 94% and 99%, correspondingly.

Table 17. Comparison of Purchase Smart Contract code

Source Units	**Generated**	**Original**
nSLOC	48	62
Complexity Score	40	42
AST Element	**Generated**	**Original**
Block	7	9
EmitStatement	4	4
EnumDefinition	1	1
EnumValue	4	4
EventDefinition	4	4
ExpressionStatement	18	19

FunctionCall	15	15
FunctionDefinition	5	5
ModifierDefinition	2	4
ModifierInvocation	6	8
StateVariableDeclaration	4	4
VariableDeclaration	5	6
Similarity		
SmartEmbed Similarity Score	0.944	
Cosine Similarity	0.996	
Smart Contract Function	**Generated**	**Original**
constructor+deployment	598342	727904
abort()	25123	47023
confirmPurchase()	30028	47133
confirmReceived()	29998	29998
refundSeller()	32252	32252

A more significant difference can be observed in the smart contract execution cost comparison. The generated smart contract constructor+deployment cost is reduced by 18%, and the abort() and confirmPurchase() functions differ even by 46% and 36%, correspondingly. The change can be attributed to the modifier usage, as the approach does not generate nonrecurring condition modifiers from guard constraints. This could also be noted in a lower complexity score, as, in the generated smart contract, this comes from a lower usage of modifiers.

3.1.1.3. StateMachine Smart Contract Code Generation

The last is the StateMachine smart contract provided together with the section on common patterns of the Solidity documentation [126] which illustrates the use of a common state machine pattern. The original smart contract supports the atState modifier and also outlines an approach for implementing the automatic time executions by using the timedTransitions modifier which is normally unsupported in Solidity.

The same as outlined previously, the smart contract was specified by using the class diagram (**Figure 46**, left) employing the Blockchain PIM profile. The structure includes 5 operations (bid(), reveal(), g(), h(), i()), and the naming is based on the provided Solidity documentation implementation. The main smart contract logic is outlined by using the state machine diagram (**Figure 46**, right), specifically, five states are outlined. Between the states, there exist 2 Transitions with Time Events, and five Call Event Operations. The Time Events specification for execution cost testing purposes is specified by using 2-minute and 3-minute intervals instead of the original 10 days and 2 days provided in the documentation.

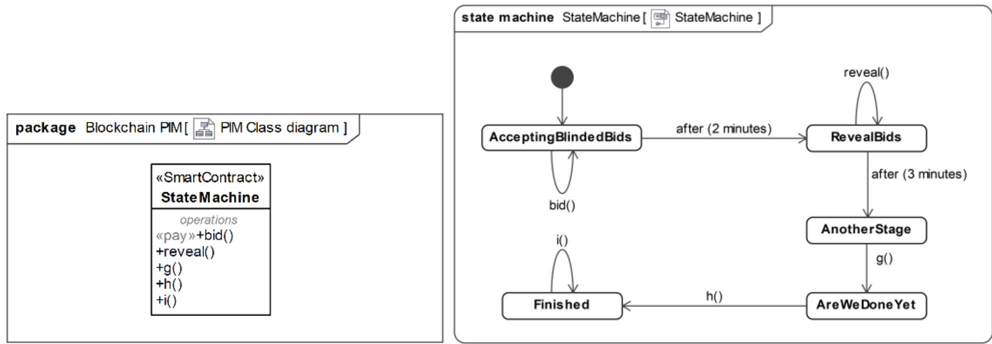


Figure 46. Blockchain PIM StateMachine smart contract

After the specification, the transformation of the Blockchain PIM to the Ethereum PSM is performed. The resulting smart contract structure and behaviour of the StateMachine are presented in **Figure 47**. The structure of the Ethereum PSM compared to the Blockchain PIM now includes the state and the creationTime variable. The state variable, State enum, and its literals are appended by using the same principle as defined previously, the creationTime is added based on the Time Event usage, and it serves as a record defining the specific timestamp when the smart contract was deployed. Additionally, two modifiers are generated and applied to all functions, atState because of the function usage as Call Events, and the timedTransitions modifier applies to all functions based on the Time Event usage.

The state machine behaviour specification between the Blockchain PIM and the Ethereum PSM differs, which is entirely based on the timedTransitions modifier implementation. Specifically, any relative time event triggers are changed into [block.timestamp >= creationTime + 2 minutes] guard constraints for evaluating the current time relation with the timed transitions; additionally, the subsequent events are added up together. The logic of transitions between the states, based on the transformed time events and Time Expressions, is recorded in the TimedTransitions modifier applied to all the Ethereum PSM smart contract functions (**Figure 47**).

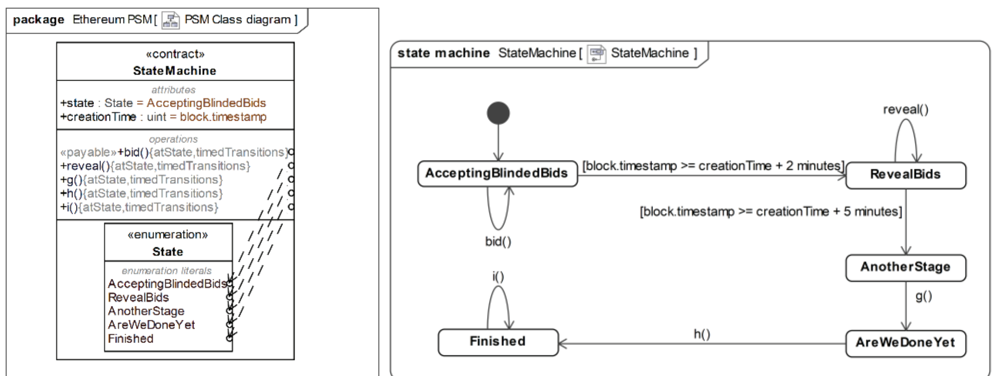


Figure 47. Ethereum PSM StateMachine smart contract

The Ethereum PSM StateMachine was transformed into the Solidity smart contract code and compared to the original. The comparison results are presented in **Table 18**. The generated smart contract code is lower by 17 nSLOC and 7 points of the complexity score. The difference is due to the removal of the transitionNext and the nextState function modifiers which are not included in the generated smart contract code. The transitionNext and nextState are unsupported because the implementation is relevant only in a few cases, and the same results are achieved by using the state declarations in the corresponding function code.

The difference of the generated and original is also reflected in the AST element count, namely, the Block, Modifier Invocation, while the most notable difference is the FunctionCall element count. The nextStage function is used repeatedly in the original smart contract. Regardless of the difference, the StateMachine achieves the same results by using the automatically generated unit tests. Furthermore, the calculated SmartEmbed and cosine similarity scores reach 82% and 95%, correspondingly.

Table 18. Comparison of StateMachine Smart Contract

Source Units	**Generated**	**Original**
nSLOC	29	46
Complexity Score	29	36
AST Element	**Generated**	**Original**
Block	7	9
EnumDefinition	1	1
EnumValue	5	5
ExpressionStatement	7	8
FunctionCall	1	6
FunctionDefinition	5	6
ModifierDefinition	2	3
ModifierInvocation	10	12
StateVariableDeclaration	2	2
VariableDeclaration	3	3
Similarity		
SmartEmbed Similarity Score	0.827	
Cosine Similarity	0.951	
Smart Contract Function	**Generated**	**Original**
constructor+deployment	596266	622427
bid()	26167	26167
reveal()	26213	26213
g()	29608	30414
h()	27015	27418
i()	23949	23949

Although smart contracts differ in terms of the code metrics, the same difference cannot be observed at the function execution level. The gas cost of the generated smart

contract deployment is lower, which is based on the complexity of the smart contract. Otherwise, the execution costs of the functions are negligible; they are around 3%.

It should be noted that the MDA-based method for smart contract-based system development is capable of generating the executable Ethereum platform Solidity smart contract code.

3.1.2. Hyperledger Fabric Go Chaincodes

For evaluating the model-to-model and the model-to-text transformation tailored for the Hyperledger Fabric, a chaincode AssetTransfer example from the Hyperledger documentation was selected. Unfortunately, no smart contracts were found that could be used for a comparison including state machine-like behaviour, so the Hyperledger Fabric section is tailored more towards the behaviour specification using the Sequence diagrams.

The developed Blockchain PIM was then exported as an XMI file and imported to the Eclipse ATL tool, and then used for model-to-model transformation. The Hyperledger Fabric PSM was then extended by specifying the function behaviour while using sequence diagrams, and the defined PSM was then used for the Go chaincode code production using the templates implemented in the Eclipse Acceleo tool.

Lastly, the evaluation of the generated AssetTransfer source code and the original is presented at the end of the section comparing the original and the generated chaincode. The Hyperledger Fabric Go chaincode is compared by the lines, cyclomatic complexity in terms of the code metrics, and, additionally, by the execution of the chaincode functions in terms of time.

3.1.2.1. AssetTransfer Smart Contract

The selected chaincode AssetTransfer in the Hyperledger Fabric documentation [119] is provided as an example to represent simple ledger query functions, such as creating, updating, or deleting assets. By using the provided chaincode, a smart contract structure (**Figure 48**) was specified by using the blockchain PIM definition rules. The smart contract itself consists of seven operations for creating, reading, updating, deleting, and transferring assets, and one utility operation for populating the ledger with data; additionally, a containing Asset Class to represent the data structure was specified, and it consisted of five Properties.

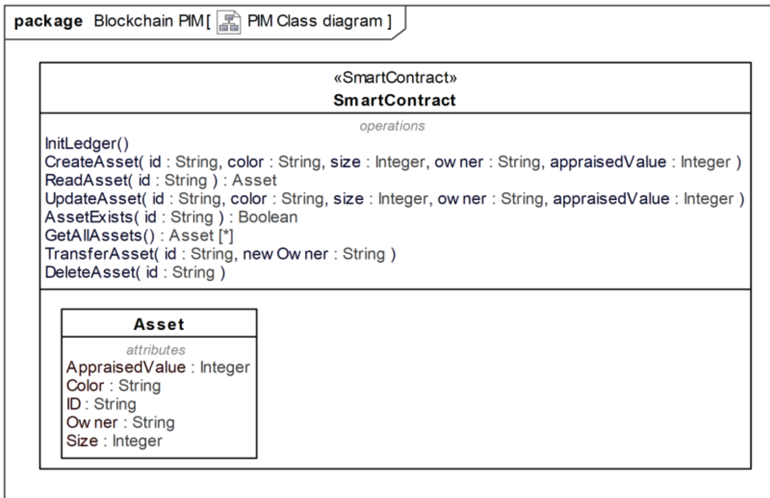


Figure 48. Blockchain PIM SmartContract contract: Class diagram

Since the smart contract does not support any behavioural logic that could be relocated to the state machine, the Blockchain PIM Smart contract specification was exported as an XMI file and transformed into the Hyperledger Fabric PSM. The resulting Hyperledger Fabric PSM SmartContract chaincode structure is represented in **Figure 49**. During the transformation, the Hyperledger Fabric PSM UML profile stereotypes were applied to Operations, Classes, and Properties. Additionally, the Go-specific types were applied instead of the UML Primitive types; beside that, each function parameters were appended with ctx TransactionContextInterface and err error Parameters.

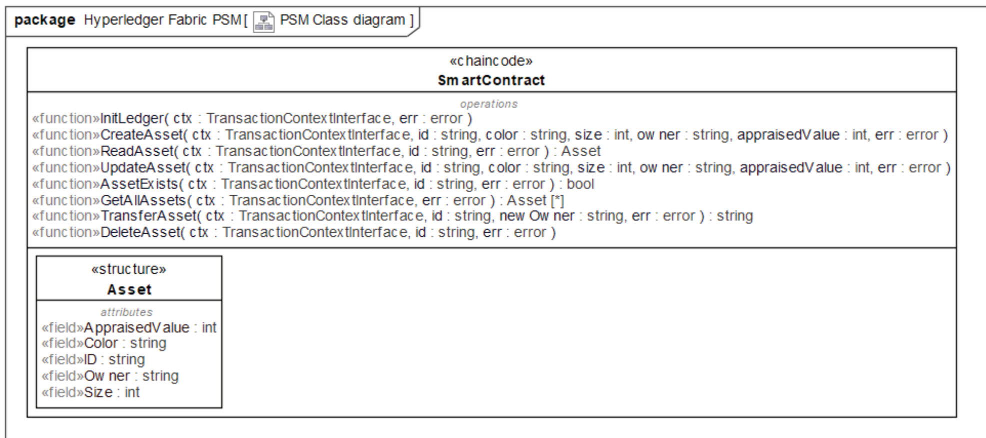


Figure 49. Hyperledger Fabric PSM SmartContract chaincode: Class diagram

Following the specification, the sequence diagrams for the chaincode functions can be specified. The InitLedger is an auxiliary function for testing purposes which adds a base set of Assets to the ledger and which was not specified as an interaction.

The InitLedger code was embedded as an opaque behavior to the chaincode specification.

The AssetExists function (**Figure 50**) starts by calling a TransactionContextInterface function to query the data from the ChaincodeStub. The GetState returns a byte assetJSON array and checks for the byte array contents, and, if the data are populated, it returns a bool value of true.

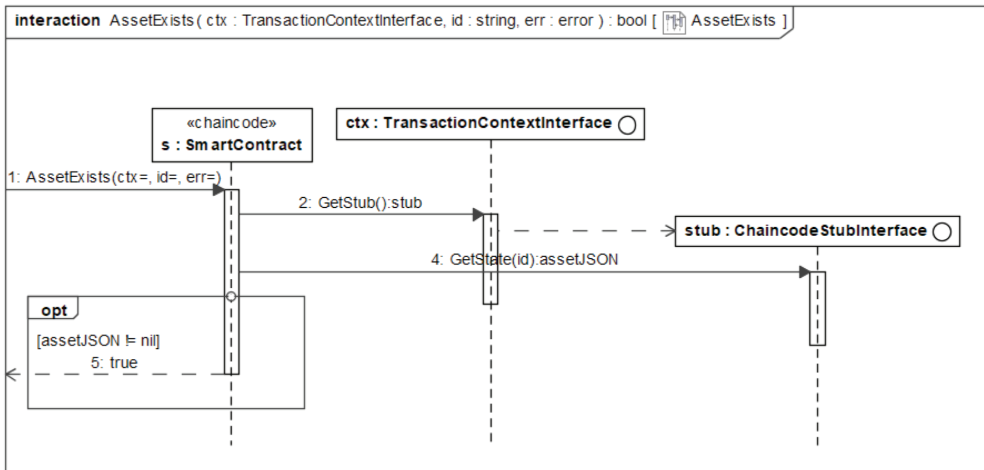


Figure 50. Hyperledger Fabric PSM SmartContract chaincode AssetExists function: Sequence diagram

The CreateAsset function (**Figure 51**) checks whether an asset exists by querying the AssetExits function; if an error is not returned and the asset does not exist, an asset structure is created and populated with data. Then, by using the default library, a byte array assetJSON is created and passed to the stub interface for data insertion.

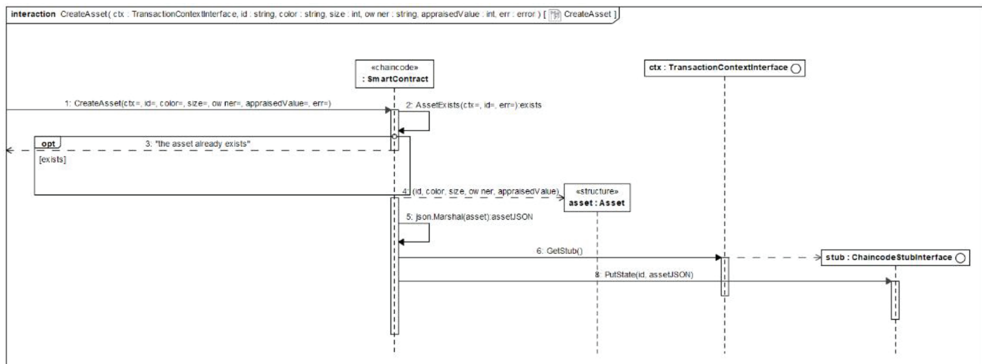


Figure 51. Hyperledger Fabric PSM SmartContract chaincode CreateAsset function: Sequence diagram

The ReadAsset function (**Figure 52**) starts by calling TransactionContextInterface to select the stub interface and by querying the data

from the ledger. If an unexpected error does not occur, the asset data are returned by populating the data structure from the selected assetJSON byte array.

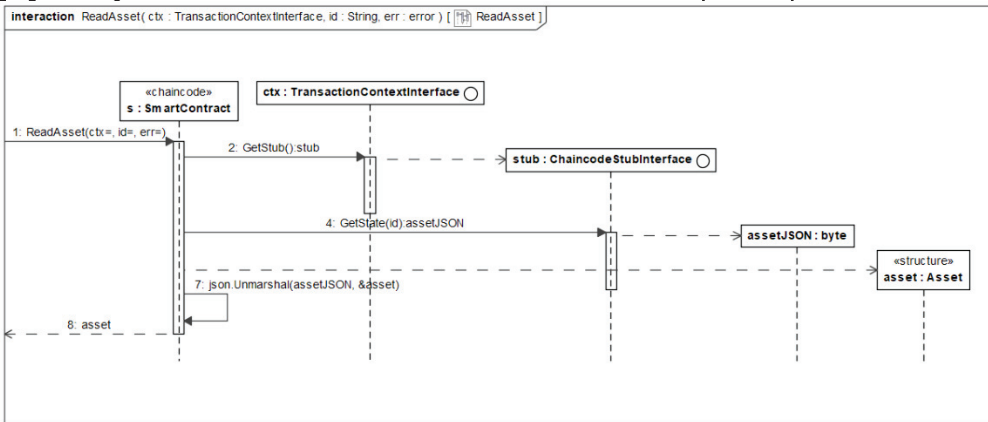


Figure 52. Hyperledger Fabric PSM SmartContract chaincode ReadAsset function: Sequence diagram

The UpdateAsset function (**Figure 53**) checks whether the asset exists by calling the AssetExists function, and again, an error is printed by using the default library. Otherwise, a new asset data structure is created which is then transformed into the assetJSON byte array and passed to the Chaincode stub for insertion.

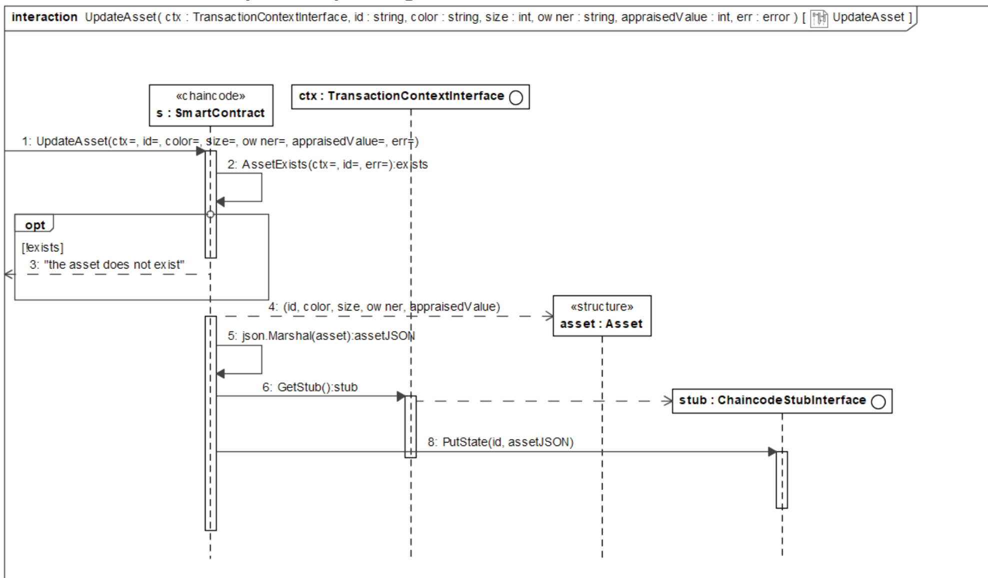


Figure 53. Hyperledger Fabric PSM SmartContract chaincode UpdateAsset function: Sequence diagram

The DeleteAsset function (**Figure 54**) using the AssetExists function checks whether an asset exists, and, if it does, the stub chaincodestubinterface deletes the record based on the passed ID.

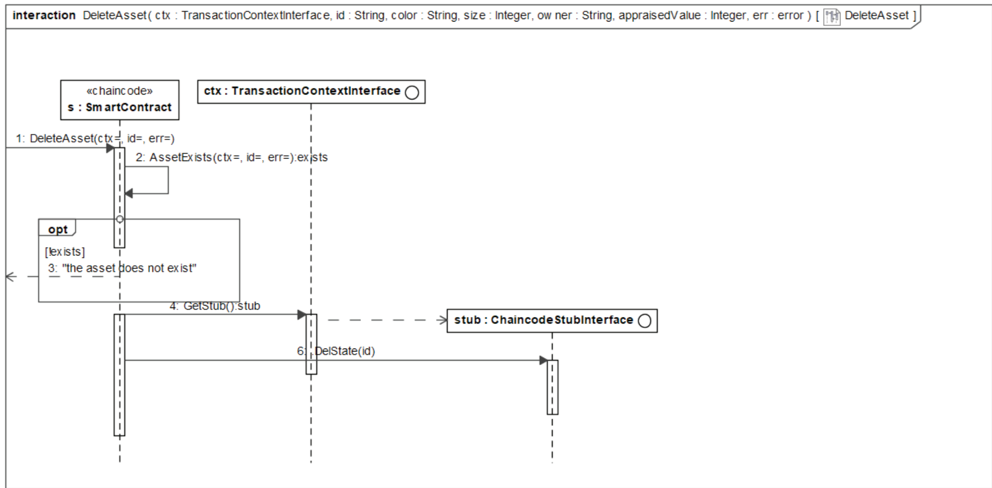


Figure 54. Hyperledger Fabric PSM SmartContract chaincode DeleteAsset function: Sequence diagram

The TransferAsset function (**Figure 55**) works similarly to the update function, but, instead of updating the full data of an asset, the update deals mainly with the change of the property of the owner. First, the asset data is selected by calling the ReadAsset function. The new string variable is created and assigned a value of asset.Owner. Then, asset.Owner is changed to the newOwner, and the data are transformed into bytes, and, by querying, the stub is appended to the ledger. The function also returns an oldOwner string.

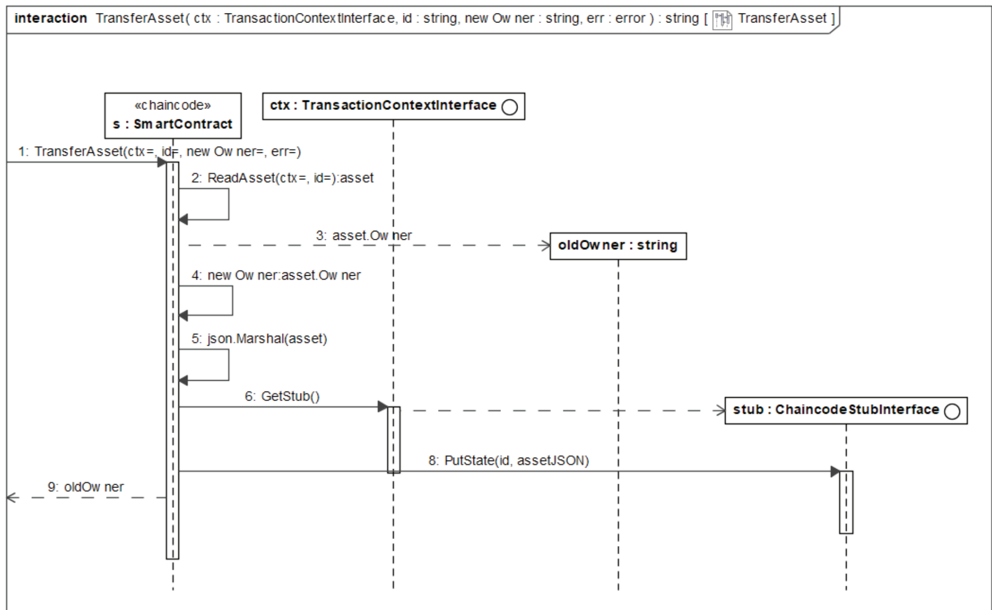


Figure 55. Hyperledger Fabric PSM SmartContract chaincode TransferAsset function: Sequence diagram

The GetAllAssets function (**Figure 56**) queries the stub to select the full range of records and creates a range iterator. This range iterator together with the loop fragment specifies a sort of while loop, and, while the next record exists in the resultIterator, the query response is returned and unmarshalled to the asset variable. The asset variable is then appended to the assets array. Once the loop is done, the function returns an array of assets.

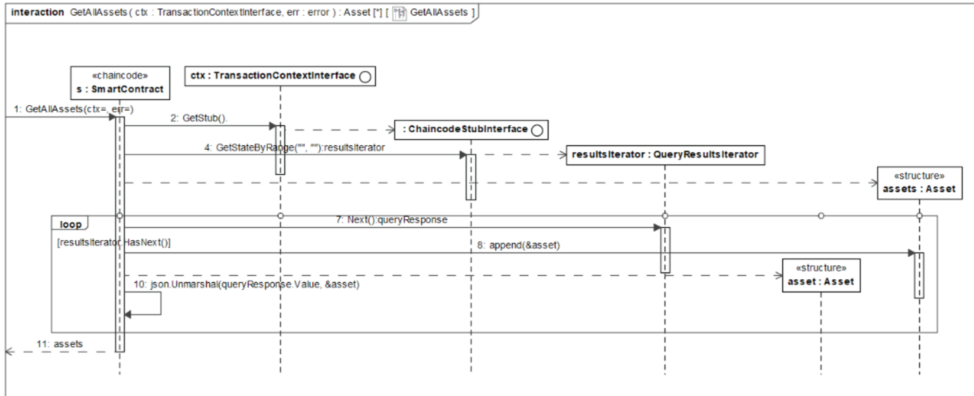


Figure 56. Hyperledger Fabric PSM SmartContract chaincode GetAllAssets function: Sequence diagram

After the Sequence Diagram definition, the Hyperledger Fabric PSM specification was exported as an XMI file, and, by using the MOFM2T transformation templates, the chaincode code. The generated chaincode code was compared to the original code in terms of the cyclomatic complexity, the lines of code, and the parameter count calculated by using the VSCode plugin. Additionally, the generated and original smart contracts were deployed on a Hyperledger Fabric network and used to calculate the execution time of each function.

Table 19. Comparison of SmartContract chaincode code

Code Metrics	**Generated**			**Original**		
Function	Cyclomatic Complexity	Lines of Code	Parameter Count	Cyclomatic Complexity	Lines of Code	Parameter Count
InitLedger	4	21	1	4	21	1
AssetExists	2	8	2	2	7	2
CreateAsset	4	16	6	4	21	6
ReadAsset	4	16	2	4	15	2
UpdateAsset	4	16	6	4	21	6
DeleteAsset	3	11	2	3	10	2
TransferAsset	4	18	3	4	17	3
GetAllAssets	5	22	1	5	21	1

Execution time (in seconds)		
Function	**Generated**	**Original**
InitLedger	2,322	2,323
GetAllAssets	0,033	0,031

CreateAsset	2,139	2,139
ReadAsset	0,018	0,018
AssetExists	0,009	0,009
TransferAsset	2,137	2,133
DeleteAsset	0,015	0,013

In terms of the Code metrics, the generated and the original versions hardly differ at all, and the main cyclomatic complexity and parameter count are the same throughout all the functions, and only the lines of the code for each function appear to vary. This is because the method is not able to produce multi-function calls in a single line; therefore, in some cases, the lines of the code have increased, but still, the functionality remains the same.

Furthermore, each chaincode function, except for the InitLedger function, was executed 50 times, and the average execution time in milliseconds barely differs. The most notable change is for the DeleteAsset function, where the average execution time changed by ~10%, and the GetAllAssets function execution time increased by ~6%; still, the difference in both cases is about 2 milliseconds.

Based on the outlined transformations to Hyperledger Fabric, it can be concluded that the MDA-based method is capable of generating an executable Go programming language code. Additionally, the produced chaincode code hardly differs from the original it is based on while outlining that the Sequence diagrams can be used for the specification of the function behaviours. Unfortunately, the specified sequence diagrams need to be specified at a level closely resembling the code; this requires considerable investment for such specifications.

3.2. MDA-based method for smart contract-based system development evaluation

For the proposed MDA-based method for smart contract-based system development evaluation, the development of a solution for hackathon certificate issuing following the method guidelines was performed. The artefacts produced during the Blockchain CIM, PIM and PSM definition and the transformation results between different models are provided further in this section.

The section consists mainly of the evaluation of model transformations listing the quantitative model-to-model transformation results for the Blockchain PIM, the Ethereum PSM, and the Hyperledger Fabric PSM. Additionally, an evaluation of the produced smart contract artefacts for both the Solidity and the Go programming languages is presented as well. The results of the experiment were also published in a paper [127].

3.2.1. Blockchain CIM Definition

The business process modelling, while supported, is neither constrained by any rules of the proposed method, nor does it provide the guidelines for the business process model development. Regardless of the business process model outlining for issuing the certificate, solution submission in a hackathon was outlined. The main part of the development solution deals with the certificate issuing process using blockchain. As a result, UML activity diagrams for issuing certificates (**Figure 57**)

and solution evaluation (**Figure 58**) business processes were outlined as a basis for the subsequent specification of the model.

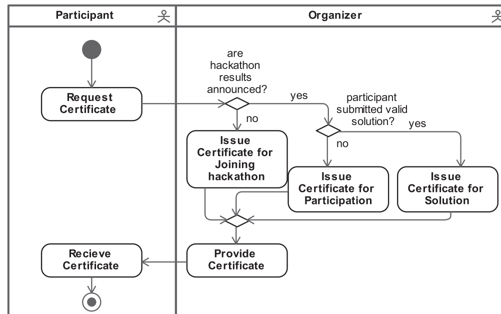


Figure 57. Blockchain CIM Issue certificate business process: Activity diagram

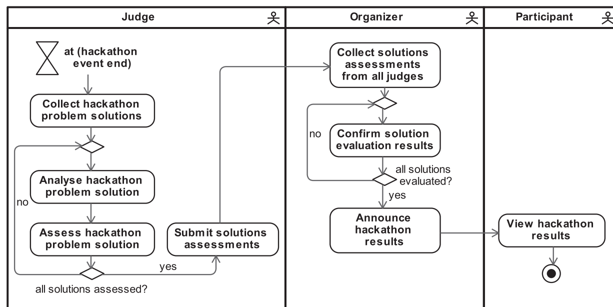


Figure 58. Blockchain CIM Evaluate submitted solutions business process: Activity diagram

Regarding the defined business model, a use case diagram was specified which outlines the hackathon organizer's use cases (**Figure 59**).

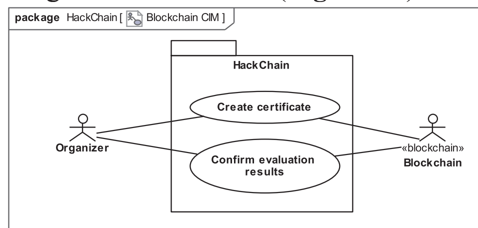


Figure 59. Blockchain CIM HackChain use case model: Use case diagram

In addition, a domain model was outlined consisting of the entity domain classes that outline the relevant domain concepts of the hackathon organization. The participant may submit a solution which is evaluated by hackathon judges, and, based on the participation, receives a certificate. The class diagram outlines the domain model, and the classes relevant for the hackathon are presented in **Figure 60**.

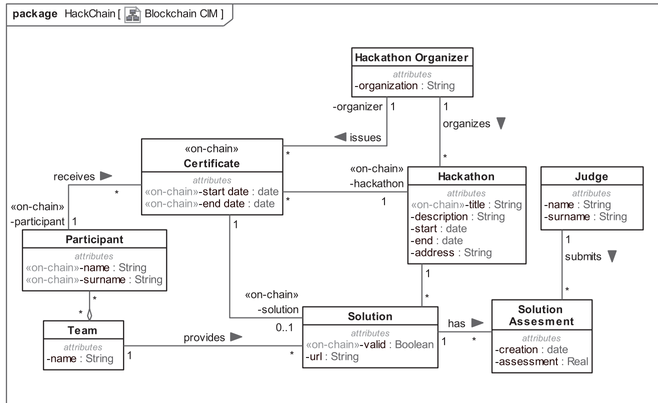


Figure 60. Blockchain CIM HackChain domain model: class diagram

The specified Blockchain CIM, containing the business, use case as well as the domain models, was modelled by using the Magicdraw CASE tool and exported as an XMI file. The XMI file that uses the implemented model to model the transformations in the Eclipse ATL tool generated the Blockchain PIM. The transformation results are provided in the following section.

3.2.2. Blockchain CIM to Blockchain PIM transformation

The results of the M2M Blockchain CIM to the Blockchain PIM transformation are presented in **Figure 61**. During the transformation, a «SmartContract» class was created, and two Operations based on the Use Cases related to the «blockchain» Actor were created. Additionally, a Blockchain CIM domain model classes and Properties specified by using an «on-chain» stereotype were transformed into a contained class.

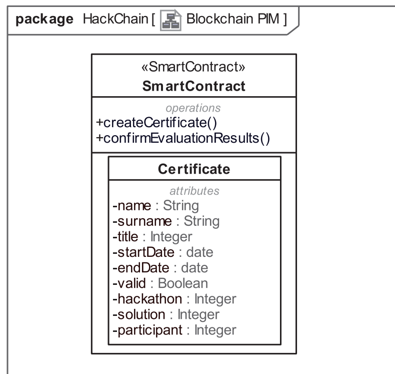


Figure 61. Blockchain CIM to Blockchain PIM Transformation result: Class diagram

The quantitative results of the model-to-model transformation are presented in **Table 20**. The provided results outline the UML model element count which is directly utilised during the transformation.

Table 20. Blockchain CIM to Blockchain PIM transformation results

Blockchain CIM	
Element	Element Count
Use Case model	
Actors	2
«blockchain» Actor	1
Use Cases	2
Association	4
Domain class model	
Class	6
Property	13
«on-chain» Property	6
Association	8
«on-chain» Association member end	4
Blockchain PIM	
Element	Element Count
Smart contract structure	
Class	2
«SmartContract» Class	1
Operation	2
Operation Parameter	0
Property	9

Directly after the transformation, the Blockchain PIM includes a single «SmartContract» Class with two Operations that directly transformed the Use Cases from CIM. Additionally, a single contained Class is also created, encompassing the Properties that were specified in CIM. When considering the main structural model elements (classes, properties, and operations), in Blockchain PIM, 14 out of 18 elements were produced during M2M transformation.

The main Blockchain PIM definition activities are supported by the proposed Blockchain PIM UML profile and by the information that was specified in the Blockchain CIM. The extension of the PIM is based on information in the CIM-level business process model or is redefined based on the developer's preference. Since any further model-to-model transformation for extracting information from business processes cannot be implemented automatically, this Blockchain PIM extension is performed manually. Following this, the Blockchain PIM was extended by appending two Operations, Operations Parameters, editing the Certificate containing Class and renaming the «SmartContract» Class (**Figure 62**).

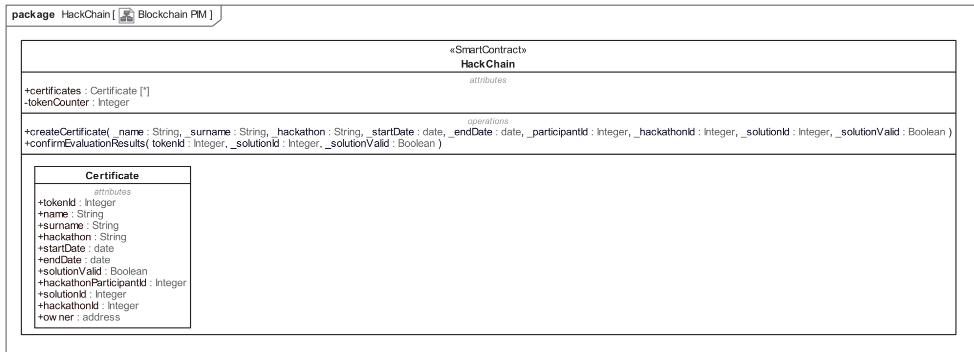


Figure 62. Blockchain PIM HackChain smart contract: Class diagram

Additionally, smart contract behaviour was specified by using the state machine diagram (**Figure 63**) based on the business processes defined in the Blockchain CIM. The specified diagram outlines the certificate transition between states based on the participant-submitted solution. The specified state machine diagram outlines the Transitions between certificate states using the specified smart contract operations as Call Event Triggers.

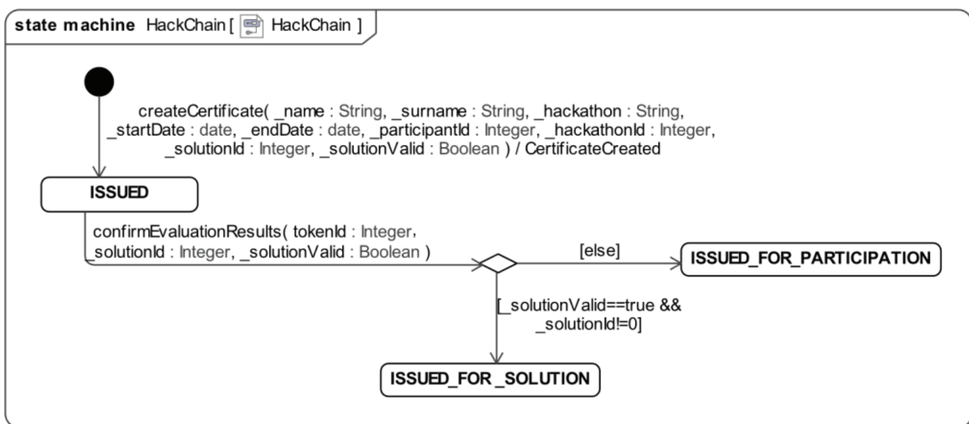


Figure 63. Blockchain PIM HackChain smart contract: State Machine Diagram

The outlined blockchain PIM encompassing the smart contract structure (**Figure 62**) and behaviour (**Figure 63**) was exported and used for transformations to the Ethereum PSM and the Hyperledger Fabric PSM.

3.2.3. Blockchain PIM to Blockchain PSM Transformation Results

The Blockchain PIM was transformed to both the Ethereum PSM and the Hyperledger Fabric PSM, and, in both cases, the redefined Blockchain PIM was exported, and, by using the M2M transformations, was transformed into Blockchain PSMs.

3.2.3.1. Blockchain PIM to Ethereum PSM Transformation Results

Specifically, during the Blockchain PIM to the Ethereum PSM transformation based on the specified State Machine behaviour, the smart contract structure was extended. The transformed Ethereum PSM is presented in **Figure 64**, and the results of the transformation of the quantitative model are presented in **Table 21**.

Table 21. Blockchain PIM to Ethereum PSM Transformation Results

Blockchain PIM	
Element	Element Count
Smart contract structure	
Class	2
«SmartContract» Class	1
Operation	2
Operation Parameter	12
Property	13
Smart contract behaviour	
State	3
Transition	4
Effect	1
Guard	2
Ethereum PSM	
Element	Element Count
Smart contract structure	
«contract»	1
«variable»	2
«mapping»	1
«struct»	1
«member»	12
«enum»	1
«function»	2
«parameter»	12
«event»	1
Smart contract behaviour	
State	3
Transition	4
Effect	1
Guard	2

The Ethereum PSM «contract» class was directly produced from a «SmartContract» Class in the PIM model; additionally, the smart contract «struct» class was generated from the contained Class; besides, a «variable» property and «enum» were created. Additionally, the operations had a «function» stereotype outlined. The transformed Ethereum PSM differs from the Blockchain PIM in terms of the generated CertificateState «enum» Enumeration, and Enumeration Literals

based on the specified State Machine States (ISSUED, ISSUED_FOR_PARTICIPATION, ISSUED_FOR_SUBMMISION), and, based on the specified Effect, a «function» Operation was generated as well.

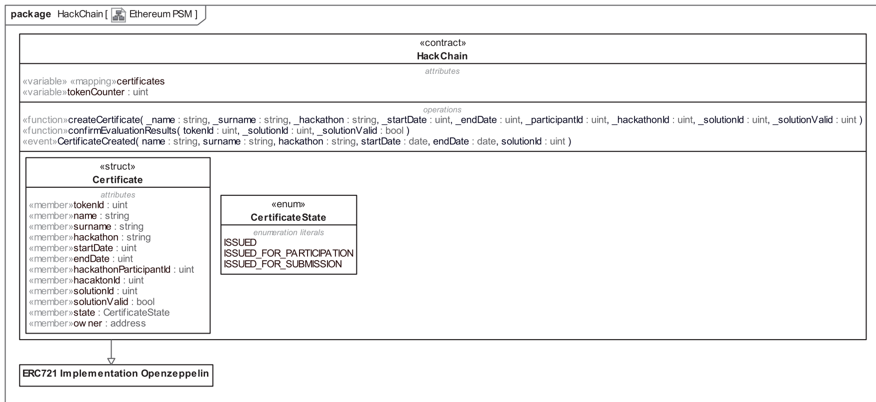


Figure 64. Ethereum PSM HackChain smart contract: Class diagram

The Ethereum PSM smart contract was specified to inherit the functionality of the ERC 721 standard implementation which is specified as a generalization between classes; additionally, «function» behaviour implementations were selected from the Opaque Behavior library.

The specified Ethereum PSM state machine diagram compared to Blockchain PIM is basically the same, as only the «function» Operation Call Event specification differs.

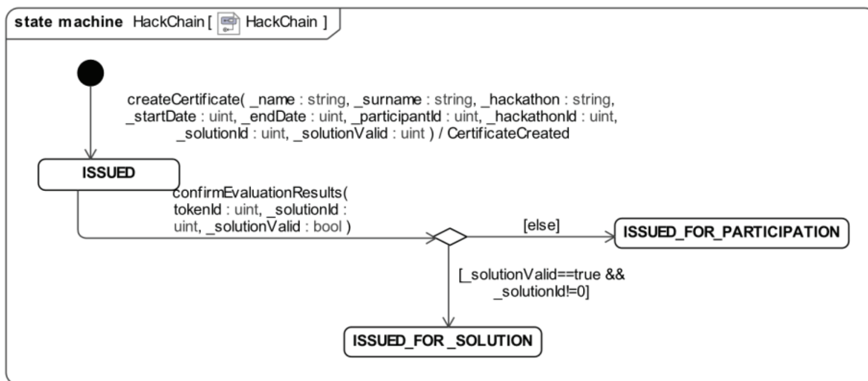


Figure 65. Ethereum PSM HackChain smart contract: State Machine Diagram

The specified Ethereum PSM, which encompasses the smart contract structure and behaviour specifications, was used as input for M2T transformations.

3.2.3.2. Blockchain PIM to Hyperledger Fabric PSM Transformation Result

Similarly, as in Ethereum PSM, the blockchain PIM was also transformed into the Hyperledger Fabric PSM. The results of the transformation from model to model are presented in **Table 22**.

Table 22. Blockchain PIM to Hyperledger Fabric PSM Transformation Results

Blockchain PIM	
Element	Element Count
Smart contract structure	
Class	2
«SmartContract» Class	1
Operation	2
Operation Parameter	12
Property	13
Smart contract behaviour	
State	3
Transition	4
Effect	1
Guard	2
Hyperledger Fabric PSM	
Element	Element Count
Smart contract structure	
«chaincode»	1
«variable»	1
«structure»	2
«field»	18
«function»	2
«argument»	16
Smart contract behaviour	
State	3
Transition	4
Effect	1
Guard	2

The Hyperledger Fabric PSM «chaincode» Class was directly produced from a «SmartContract» Class in Blockchain PIM; additionally, the smart contract «structure» Class was generated from the contained Class and the specified effect in the Blockchain PIM state machine diagram. Additionally, the operations had a «function» stereotype outlined. Still, the most notable difference is the addition of ctx TransactionContextInterface and err error Parameters to each «function».

Additionally, the remaining stereotypes were applied, and new data types were updated for each parameter. Differently from the Ethereum PSM, the state Parameter was transformed into the string «field» and the certificate «structure» Class, and the naming changed based on the denoted visibility in the Blockchain PIM model. The Hyperledger Fabric PSM is made up of a Class diagram (**Figure 66**) which describes the smart contract structure, while the behaviour is presented in **Figure 67**.

Similarly to the Ethereum PSM, the Hyperledger Fabric «chaincode» Class was specified to inherit the functionality of the ERC 721 standard implementation.

Additionally, two «function» behaviour implementations were selected from the Opaque Behavior library.

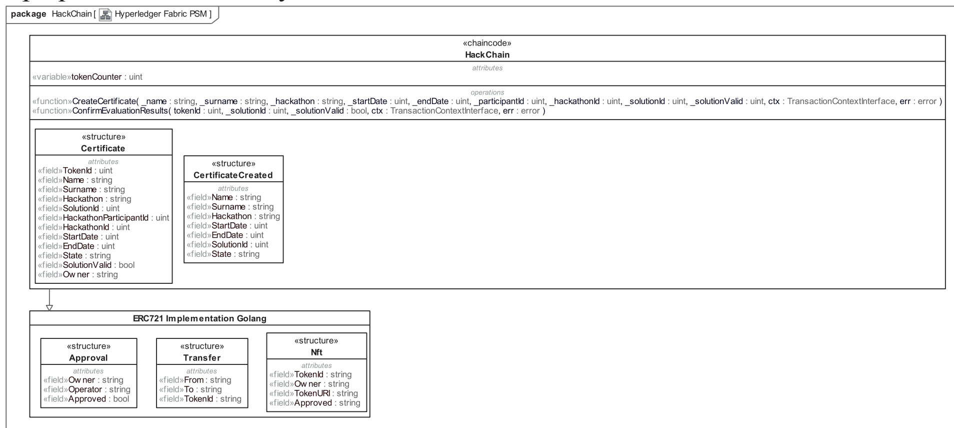


Figure 66. Hyperledger Fabric PSM HackChain smart contract: Class diagram

Furthermore, the state machine was updated as well, like in the Ethereum PSM. Compared to the Blockchain PIM, the Hyperledger Fabric PSM State Machine Transitions have updated Call Event Operations, whereas other elements are the same.

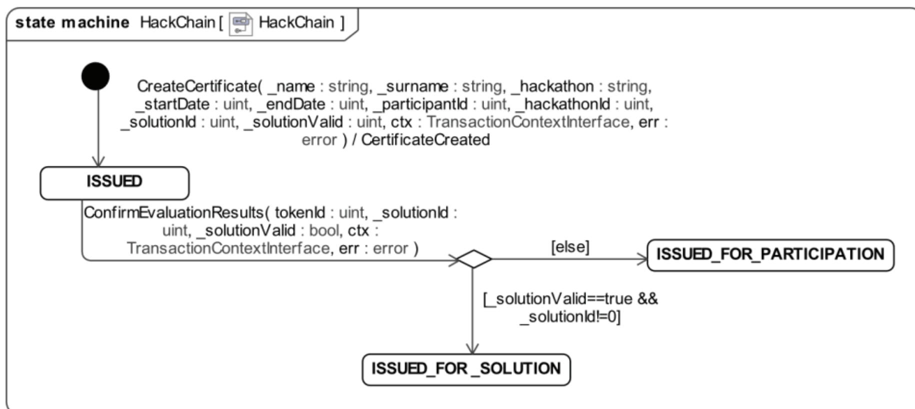


Figure 67. Hyperledger Fabric PSM HackChain smart contract: State Machine Diagram

The Hyperledger Fabric PSM, encompassing the smart contract structure and behaviour specifications, was used as input for M2T transformations. Just like the Ethereum PSM, the Hyperledger PSM was exported as an XMI file and was used during the MOFM2T implementations in the Eclipse Acceleo tool.

3.2.4. Blockchain PSM to code transformation results

After the Blockchain PSM specification, both the Ethereum PSM and the Hyperledger Fabric PSM were used to produce smart contracts. In order to evaluate whether generated smart contracts meet the requirements, the Go chaincode and the

Solidity smart contract have been tested by using the same test scenarios. Furthermore, the produced smart contracts were test units using the ERC721 standard-provided smart contract unit tests (https://github.com/m-jurgelaitis/MDAsmartCD).

A smart contract code was generated, and code metrics are presented in **Table 23**. The results overview the HackChain smart contractwhich includes the createCertificate and the confirmEvaluationResults behaviour. In both cases, the generated smart contract was evaluated only by including the smart contract code which is generated during the transformation. This is particularly relevant in the chaincode case, as the Go programming language does not support inheritance, so the ERC 721 standard implementation was directly included in the same chaincode file.

Table 23. HackChain Solidity smart contract code metrics

HackChain Solidity smart contract				
Function	**NLOC**	**CCN**	**Token**	**Parameters**
createCertificate	7	1	109	9
confirmEvaluationResults	10	4	74	3
HackChain Go chaincode				
Function	**NLOC**	**CCN**	**Token**	**Parameters**
CreateCertificate	16	3	154	10
ConfirmEvaluationResults	24	7	142	4

The generated code was analysed for evaluating the code quality in terms of security and complexity. Additionally, static code analysis tools were used to analyse the two generated smart contracts code. Slither [128], a static code analysis tool used for automated vulnerability identification, optimization analysis, code understanding analysis, and assisted code review, was used to assess the generated Solidity HackChain smart contract. No high, medium, or low issues were discovered in the smart contract out of 80 detectors that the Slither tool checks. Correspondingly, the Go chaincode was evaluated by using a framework for detecting vulnerabilities of the Hyperledger Fabric smart contracts. HFCCT [129], a tool for evaluating the Hyperledger Fabric smart contracts, was used to evaluate common vulnerabilities. Out of 17 types of vulnerabilities, no issues were detected during the HackChain Go chaincode evaluation.

The presently listed results only outline the smart contract metrics; therefore, additionally, the smart contracts were deployed and tested on their respective networks.

3.2.5. Execution of smart contract code

The same data sets were used to run smart contract functions in the same workflow and to record the execution metrics. In both cases, 18 certificates were created (state machine state ISSUED) for the hackathon participants. In order to cover every state machine transition, the data set included data that would change the certificate state accordingly:

- It was confirmed that 9 certificates had submitted valid solutions, solution references had been updated and certificates had been updated to ISSUED_FOR_SOLUTION state.
- 6 certificates were updated to the ISSUED_FOR_PARTICIPATION state, and it was determined that the participant had presented an invalid solution.
- The remaining 3 certificates were updated to the ISSUED_FOR_PARTICIPATION state, which marks that the participant did not submit a solution at all.

The solidity smart contract was hosted on a test network to determine the gas execution costs. The evaluation of this process is presented in **Table 24**. Additionally, the generated hosted smart contract was experimented by imitating the Hackathon certificate issuing and state changes.

Table 24. HackChain Solidity smart contract gas costs

Executed Function	Transaction Count	Transaction Fee (ETH)	GAS usage	Average GAS usage
Deployment	1	0.03998626	2 843 533	2 843 533
createCertificate	18	0.07700459	4 931 184	273 955
confirmEvaluationResults	18	0.01756643	845 160	46 953

In total, 18 recorded transactions were performed, and the total amount spent in transaction fees for the createCertificate function was around 0.07700459, and it scored 0.01756643 for confirmEvaluation results.

The generated HackChain chaincode Go was also hosted on a blockchain network consisting of two organizations (peers). The chaincode was similarly tested by imitating the creation of certificates and changing the certificate state actions, the statistics of which were taken from the produced Docker log.

An average time for each function is presented in **Table 25**, the execution time encompasses how much time it takes to validate a block, to commit the block and private data to storage, and ultimately commit a block to the blockchain. The execution time includes only block commitment times which were made up of a single transaction and are measured in milliseconds.

Table 25. Execution of the HackChain Go chaincode (ms)

Executed Function	Transaction Count	Execution Time (ms)
Deployment	3	427
CreateCertificate	18	1715
ConfirmEvaluationResults	18	1583

In total, 18 transactions were recorded during testing with an average time of 1583 milliseconds for the ConfirmResults function, and 1715 milliseconds for the CreateCertificate function. Additionally, it should be noted that the state commit takes the highest amount of time – as expected – as it commits changes to every single peer database.

The presented results indicate that the MDA-based method can be used to automate the code generation from the specified models. The experiment also demonstrates that the specified models, once transformed into PSMs, can support multiple platforms, and the generated smart contracts can be utilised in a smart contract-based system development.

3.3. MDA-based method for development of smart contract-based systems comparison

The proposed method for smart contract-based system development can be directly compared to other MDA-based approaches. The comparison of the proposed method and the approaches analysed previously are presented in **Table 26**. The approaches are compared in terms of the CIM, PIM, and PSM contents, and the object is modelled or specified at the abstraction layer. The notations that are used for the model specification are determined, and the implementation of model transformation is considered, at each layer, and lastly, the supported platforms are outlined.

Table 26. MDA-based approaches comparison for smart contract-based system development

Method Abstraction Layer		Smart Contract Engineering [71]	A model-driven approach to smart contract development [91]	The Development of Smart Contracts for Heterogeneous Blockchains [89]	[93] MDE4BBIS	Proposed method
CIM	Specification Object	Contract parties agreement	ADICO statements	REA ontology	Does not employ CIM	Business process model Use case model Domain model
	Specification Notation	Textual	Textual	Class diagram		UML activity diagram, UML use case diagram, UML class diagram
	Extension	-	-	UML profile		UML profile
	Model to model transformation implementation	ATL planned	Manual	Manual		ATL
PIM	Specification Object	Formal Smart Contract Description	Finite State Machine	Commitment-based ontology	Business logic model Process Model Activity Model Blockchain Technical Design Model	Smart Contract Structure Smart contract behaviour
	Specification Notation	Event-B	Finite State Machine	UML class diagram	Not defined	UML class diagram, UML state machine diagram
	Provided Extension	Not defined	-	UML profile	UML profile	UML profile
	Model to model	ATL	Manual	Manual	Manual	ATL

	transformation implementation	planned				
PSM	Specification Object	ER model	Solidity smart contract code	Commitment-based ontology	Object-Event Table Existence Dependency Graph	Smart Contract Structure Smart contract behaviour
	Employed Notation	Not defined	Code produced directly from PIM	UML class diagram	Not defined	UML class diagram, UML state machine diagram, UML sequence diagram
	Model-to-text transformation implementation	Not defined		Acceleo M2T	Not defined	Acceleo M2T
	Provided Extension	Not defined		UML profile OCL constraints	UML profile	UML profile
	Supported Platforms	Ethereum	Ethereum	Ethereum Hyperledger Fabric	Ethereum Corda Hyperledger Fabric	Ethereum Hyperledger Fabric
	Supported Programming Languages	Solidity	Solidity	Java, Solidity	Not defined	Go, Solidity

Compared to the other MDA-based approaches employing textual notations or not employing CIM at all, the proposed method in this dissertation has a clearly defined CIM abstraction layer, which, when using UML, specifies the business processes, use case and domain models. Similarly to the other approaches, the proposed method supports the PIM and PSM smart contract structure and behaviour specifications using state machines, although, for the behaviour specification, an alternative using a sequence diagram is also proposed. Additionally, support for the already outlined smart contract implementation standards is provided as well which are used as opaque behaviors directly at the PSM level. In terms of automation, when compared to the other approaches directly, the proposed method achieves a higher automation level, as only one method supports the automated model to text transformations. Additionally, in the proposed method, the validation of the model in the Blockchain CIM definition is automated, and so is the model to model transformation and validation at the Blockchain PIM and PSM abstraction level. Additionally, the proposed method automates model to text transformations at the smart contract development level, thus producing the smart contact code. Lastly, the method supports multiple blockchain technology platforms as well as code generation to multiple programming languages.

3.4. Threats to the validity of the experiment

CIM, PIM and PSMs have been developed by the author of the method and have been used for the production of a smart contract code. Generally, the method is supported by the development of a model, during which, it requires the developer to

be highly familiar with modelling as a poorly designed model cannot ensure that the smart contract code will be executable.

During the experimental evaluation, the method was demonstrated to be capable of generating the smart contract code which is used in the platform/language documentation as examples; unfortunately, between different platforms, such an evaluation differs. Considering that a wider plethora of tools exist for Solidity code analysis, more comprehensive analysis was performed for Solidity smart contracts. The Hyperledger Fabric chaincode analysis suffers in this regard, as the direct similarity analysis cannot be performed, and an approximate comparison can only be made.

Additionally, the evaluation is limited in terms of the generated code quality when evaluating the smart contracts in the HackChain solution. The currently existing static code analysis tools cannot be used to evaluate the code of both platforms. For this reason, two different static code analysis frameworks are used which evaluate a different set of vulnerabilities. The Slither tool, while comprehensive, can only be used for Solidity, and the HFCCT tool is capable of detecting only the common types of vulnerabilities and issues in the Hyperledger Go chaincode.

Direct comparison of the proposed method to other approaches is limited, mainly because the results of other approaches in terms of the model specification, transformation, and code production activities are not available for review. Following this, only a fragmented comparison between MDA-based approaches is provided, comparing the design of the method and the overall approach to the model-based smart contract development.

IV. CONCLUSIONS

1. The process of developing smart contract-based systems is highly specialised for each blockchain platform, and, during the preliminary phases of development, a considerable amount of effort is spent identifying the applicability of the technology for specific requirements, which is further complicated by the lack of well-established methods for smart contract development. The analysis shows that the utilisation of modelling can be used to support smart contract-based system development activities ranging from the requirement specification, the system design, or even the automating code generation utilising model transformations prevalent in the Model Driven Architecture.
2. The analysis of the application of MDA-based approaches for the specification of the blockchain technology artefacts has shown that most of the proposals are in the conceptual stages. The analysed approaches do not fully employ all MDA abstraction layers: the CIM layer is often ignored, or it specifies requirements using only textual notations; the PIM and PSM layers do not have a clear distinction between the two; the PSM layer is sometimes ignored, and the code is generated from PIM. Most of the approaches primarily focus on supporting the code generation and validation activities for Solidity smart contracts, with only a few offering multiplatform support. None of the analysed approaches employ the model-to-model transformations, and only some utilise model-to-text transformation for code production.
3. The model-driven development method based on the MDA for smart contract-based system development has been proposed, which includes the Blockchain CIM, the Blockchain PIM, and the Blockchain PSM abstraction layers. Based on comparative analysis of the MDA approaches, the proposed Blockchain CIM abstraction level is outlined to model the business processes and the requirements of the smart contract-based system. The Blockchain PIM abstraction level is used to model the general behaviour and structure of the smart contract without relying on platform-specific details to enable multiplatform support. The outlined Blockchain PSM is used for platform-specific smart contract structure and behaviour definition and code production. Similarly to other MDA approaches, the method is tailored for the Ethereum and Hyperledger Fabric platforms, but it enables extensions for other smart contract implementation platforms as well.
4. Blockchain CIM, PIM, and PSM UML profiles have been implemented for the proposed MDA-based method by using the Magicdraw CASE tool, thus allowing to specify Blockchain CIM, PIM, and PSM models mainly for enabling model transformations. The Blockchain CIM provides an UML profile for supporting the specification of the business processes, the requirements of a smart contract-based system, and its integration with the blockchain technology representations using the UML activity, the use case, and the class diagram. Like in other MDD approaches, the smart contract structure is specified in the UML class diagram using the provided Blockchain PIM UML profile, and, for behaviour specification, a UML state machine diagram is used. The Blockchain

PSM abstraction level is used to model the behaviour and structure of the smart contract for a specific technology platform. The blockchain PSM UML profile is provided to specify the structure of smart contracts using the UML class diagram; meanwhile, for behaviour specification UML state machine diagrams are employed, and, for function-specific behaviour definitions, platform-specific UML sequence diagrams and opaque behaviours specifications are utilised.

5. The combination of the implemented model transformations for the proposed MDA-based method demonstrates that both structural and behavioural smart contract details can be specified by using UML and the proposed extensions. The information from the higher abstraction level models can be used during model-to-model transformations to outline lower abstraction level model details. Furthermore, model-to-text transformations are tailored to the Ethereum platform Solidity and the Hyperledger Fabric platform Go programming languages and are used for platform-specific smart contract code generation.
6. An assessment of the implemented method model transformations has been performed which has demonstrated that the proposed method can be used for the modelling of the smart contract structure and behaviour using UML Class, State Machine and Sequence diagrams for automating the executable smart contract code generation. The generated code has been produced from models based on the examples outlined in the Solidity and Hyperledger Fabric documentation and evaluated in terms of the code similarity and execution. In terms of the code lines and AST elements, the generated and original Solidity smart contracts are very similar, and the calculated cosine similarity varies from 82% to 99%; the cyclomatic complexity of the generated Go chaincode functions was found to the be identical to the original, thus demonstrating that transformations can produce a smart contract code that is similar to the original and which provides the same functionality.
7. The experimental evaluation of the application of the MDA-based method for the development of smart contract-based systems has been performed by applying the method throughout the development process of the solution for hackathon certificates. The results indicate that the method implementation enables the specification of the smart contract structure and behaviour that can be used to generate multiple platform-specific models, and model transformations to executable smart contract code from the defined Ethereum and Hyperledger Fabric PSMs, thus providing multiple platform support.

SUMMARY

ĮVADAS

Motyvacija

Blokų grandinė suteikia decentralizuotą, išskirstytą duomenų bazę, kuri kartu su išmaniaisiais kontraktais gali būti naudojama veiklos procesams decentralizuoti. Ši technologija gali būti pritaikoma įvairiems programiniams sprendimams kurti, norint padidinti pasitikėjimą, įgalinti atsekamumą, skaidrumą ir mažinti priklausomybę nuo centralizuotų trečiųjų šalių. Deja, blokų grandinės technologijų pritaikymas kuriant specializuotus programinius sprendimus yra siauras, nes technologijos nėra pasiekusios pakankamo brandumo lygio.

Dabartinis išmaniųjų kontraktų technologija grindžiamų sistemų kūrimas labiausiai priklauso nuo išmaniųjų kontraktų kūrimo proceso. Pagrindinis privalumas ir trūkumas yra tai, kad blokų grandinės nepakeičiamumas, kai išmanusis kontraktas negali būti pakeistas po diegimo, reikalauja koncentruoti pastangas į ankstesnius kūrimo etapus, susijusius su išmaniųjų kontraktų ir išmaniaisiais kontraktais grindžiamų sistemų kūrimu. Taip pat kūrimo proceso neapibrėžtumas ir tai, kad formalizuoti, plačiai taikomi kūrimo metodai neegzistuoja, išmaniųjų kontraktų kūrimo ir valdymo procesą dar labiau apsunkina. Dauguma kūrimo fazių taip pat neturi apibrėžtų gairių ar standartų, tai reikalauja papildomos analizės kiekvieno etapo metu, arba kūrėjai turi pasikliauti asmenine patirtimi. Aiškiausiai apibrėžtas yra realizacijos etapas, nes dalis platformų turi bendruomeninių standartų bibliotekas, kurios teikia realizacijos šablonus, nors tai ir skatina kodo klonavimą išmaniųjų kontraktų kūrimo proceso kontekste.

Norint paspartinti ir praplėsti blokų grandinių technologijų pritaikymą, reikalingos pagalbinės priemonės programinės įrangos kūrimo etapų metu. Specializuotų sprendimų kūrimas galėtų būti palaikomas pritaikant modeliavimo praktikas, taip palengvinant blokų grandinės konceptų supratimą, suteikiant galimybes aprašyti informacines sistemas bendrine notacija, palaikyti skirtingas platformas ar netgi automatiškai sugeneruoti programinius artefaktus.

Siūlymai taikyti modeliais grindžiamus programinės įrangos kūrimo metodus išmaniųjų kontraktų technologija grindžiamų sistemų ir išmaniųjų kontraktų kūrimo kontekste egzistuoja ir yra perspektyvi tyrimo sritis. Šie metodai dažniausiai palaiko automatinio kodo generavimo ir modelių validavimo veiklas, bet nepadengia viso programinės įrangos kūrimo gyvavimo ciklo. Dažniausiai metodai yra pritaikyti *Ethereum* ir *Hyperledger Fabric* platformoms, kai modeliuojama išmaniojo kontrakto struktūra naudojant klasių diagramas bei elgsena naudojant būsenų mašinas ar panašias notacijas.

Tyrimo sritis ir objektas

Disertacijos tyrimo objektas yra išmaniųjų kontraktų technologija grindžiamų sistemų ir išmaniųjų kontraktų kūrimo procesas, įrankiai, metodai ir technologijos. Tyrimo sritis apima ir išmaniųjų kontraktų technologija grindžiamų sistemų kūrimo,

ir modeliais grindžiamų programinės įrangos kūrimo, procesų automatizavimo įrankius ir metodus.

Sprendžiama problema ir tyrimo klausimai

Dabartinis išmaniųjų kontraktų technologija grindžiamų sistemų kūrimo procesas yra probleminis, nes nepakankama parama suteikiama reikalavimų specifikavimo, projektavimo ir išmaniųjų kontraktų realizacijos veikloms. Šios veiklos galėtų būti palaikomos taikant modeliavimą, kuris galėtų palaikyti automatizuotą kodo generavimą iš apsibrėžtų modelių. Toks automatizavimas sudarytų sąlygas palengvinti išmaniųjų kontraktų technologija grindžiamų sistemų kūrimo procesą. Dėl to disertacijoje bandoma atsakyti į šiuos tyrimo klausimus:

- Ar išmaniųjų kontraktų technologija grindžiamų sistemų kūrimas gali būti automatizuotas?
- Ar modeliavimas gali būti taikomas automatizuojant išmaniųjų kontraktų technologija grindžiamų sistemų kūrimą ir jei taip, kokiu būdu?
- Ar koncepciniai išmaniųjų kontraktų technologija grindžiamų sistemų modeliai gali būti transformuojami į realizacijos artefaktus ir jei taip, kokiu būdu?
- Kokios sudėties koncepciniai išmaniųjų kontraktų technologija grindžiamų sistemų modeliai yra reikalingi vykdomiems realizacijos artefaktams sugeneruoti?

Tyrimo tikslas ir uždaviniai

Disertacijos tikslas yra išplėsti išmaniųjų kontraktų technologija grindžiamų sistemų kūrimo galimybes, suteikiant galimybę automatizuotai sugeneruoti programinį kodą iš koncepcinių išmaniųjų kontraktų technologija grindžiamų sistemų modelių. Tikslui pasiekti iškelti tokie uždaviniai:

1. Išanalizuoti blokų grandinės ir išmaniųjų kontraktų technologijas, išmaniųjų kontraktų technologija grindžiamų sistemų ir išmaniųjų kontraktų kūrimo procesus ir modeliais grindžiamos programinės įrangos kūrimą;
2. Nustatyti modeliais grindžiamų programinės įrangos kūrimo metodų sąsajas su išmaniųjų kontraktų technologijomis grindžiamų sistemų kūrimu;
3. Pasiūlyti modeliais grindžiamą programinės įrangos kūrimo metodą išmaniųjų kontraktų technologija grindžiamoms sistemoms kurti;
4. Apibrėžti blokų grandinės ir išmaniųjų kontraktų modelių transformavimo į realizacijos artefaktus taisykles;
5. Realizuoti pasiūlytą modeliais grindžiamo programinės įrangos kūrimo metodą automatizuotai išmaniųjų kontraktų technologija grindžiamoms sistemoms kurti;
6. Eksperimentiškai įvertinti modeliais grindžiamo programinės įrangos kūrimo metodo taikymą kuriant išmaniųjų kontraktų technologija grindžiamas sistemas.

Tyrimų metodika

Tyrimas buvo atliktas vadovaujantis konstruktyviojo tyrimo (angl. *Constructive Research*) metodika [1]. Tyrimo metu išskirti 7 žingsniai:

1. **Tyrimo problemos apibrėžimas.** Buvo atlikta blokų grandinės ir išmaniųjų kontraktų technologijų analizė, išmaniųjų kontraktų technologijomis grindžiamų sistemų kūrimas ir nustatyta tyrimo problema, susijusi su išmaniųjų kontraktų technologijomis grindžiamų sistemų kūrimu.
2. **Tyrimo srities apibrėžimas.** Apibrėžta modeliavimo taikymo sistemų kūrimo procesuose, metoduose sritis buvo analizuojama siekiant nustatyti tinkamas modeliavimo taikymo priemones.
3. **Dalykinės srities analizė.** Buvo atlikta lyginamoji modeliavimo taikymo išmaniųjų kontraktų technologijomis grindžiamų sistemų ir išmaniųjų kontraktų kūrimo procesuose analizė.
4. **Sprendimo koncepcija.** Pasiūlytas *Model Driven Architecture* (MDA) architektūra grindžiamas metodas išmaniųjų kontraktų technologija grindžiamoms sistemoms kurti, apimantis tris skirtingus abstrakcijos lygmenis, modelių transformacijas ir modelių kūrimo procesus.
5. **Sprendimo realizacija ir vertinimas.** MDA architektūra grindžiamas išmaniųjų kontraktų technologija grindžiamų sistemų kūrimo metodas buvo eksperimentiškai ištirtas, įvertinant automatiškai sugeneruotus išmaniųjų kontraktų artefaktus naudojant realizuotas modelių transformacijas.
6. **Eksperimentinis sprendimo taikymo vertinimas.** Įvertintas siūlymo MDA architektūra grindžiamo metodo taikymas kuriant išmaniųjų kontraktų technologija grindžiamas sistemas.
7. **Teorinės svarbos analizė.** Pasiūlytas metodas buvo palygintas su kitais MDA architektūra grindžiamais metodais, skirtais išmaniųjų kontraktų technologija grindžiamoms sistemoms ir išmaniesiems kontraktams kurti.

Ginamieji teiginiai

1. Išmaniųjų kontraktų technologija grindžiamų sistemų kūrimas gali būti automatizuotas pritaikant MDA architektūros principus reikalavimų specifikavimo, projektavimo ir išmaniųjų kontraktų realizavimo veiklose.
2. Siūlomi UML notacijos plėtiniai gali būti naudojami išmaniųjų kontraktų technologija grindžiamoms sistemoms modeliuoti: 1) specifikuoti veiklos procesus naudojant veiklos diagramas, sistemos reikalavimus naudojant panaudojimo atvejų, klasių diagramas CIM abstrakcijos lygmeniu, 2) modeliuoti išmaniųjų kontraktų struktūrines ir elgsenos savybes PIM abstrakcijos lygmeniu, naudojant klasių ir būsenų diagramas 3) ir papildomai specifikuoti funkcijų elgseną naudojant sekų diagramas ir *opaque behavior* specifikacijas PSM abstrakcijos lygmeniu. Sukurti UML plėtiniai taip pat įgalina modelių transformacijas naudoti automatiškai pereinant tarp skirtingų abstrakcijos lygių modelių; žemesnio abstrakcijos lygio modeliui sugeneruoti išplečiant išmaniojo kontrakto struktūrą, naudojamos elgsenos detalės, apibrėžtos aukštesnio abstrakcijos lygmens modelyje.
3. PSM abstrakcijos lygmens išmaniojo kontrakto struktūra ir elgsena, specifikuota naudojant UML klasių, būsenų ir sekų diagramas, gali būti naudojama automatiškai sugeneruoti išmaniojo kontrakto programinį kodą

Ethereum ir *Hyperledger Fabric* platformoms, pasitelkiant modelių transformacijas.

Mokslinis naujumas

Šio tyrimo mokslinį naujumą galima apibendrinti taip:

1. Siūlomas MDA architektūra grindžiamas metodas naudoja UML ir pasiūlytus UML plėtinius išmaniųjų kontraktų technologijomis grindžiamoms sistemoms kurti, jie leidžia specifikuoti išmaniuosius kontraktus, apimant ne tik struktūros, bet ir elgsenos aspektus;
2. Siūlomas metodas yra grindžiamas MDA principais, skirtas išmaniųjų kontraktų technologijomis grindžiamoms sistemoms kurti ir apibrėžia tris skirtingų abstrakcijų lygmenų modelius, apimančius UML veiklos, klasių, panaudojimo atvejų, būsenų, sekų diagramas, taip taikant modeliavimą reikalavimų, projektavimo ir realizacijos kūrimo etapuose.
3. Siūlomas MDA architektūra grindžiamas metodas perėjimams iš vieno abstrakcijos lygio į kitą naudoja modelis į modelis (angl. *model to model*, *M2M*) transformacijas. Taip pat modelis į tekstą (*model to text, M2T*) transformacijos naudojamos išmaniųjų kontraktų programiniam kodui sugeneruoti *Solidity* ir *Go* programavimo kalbomis.

Praktinė reikšmė

Šio tyrimo praktinė reikšmė, nors ir glaudžiai susijusi su moksliniu naujumu, gali būti apibendrinta taip:

1. Siūlomas MDA architektūra grindžiamas metodas apibrėžia bendrinį išmaniųjų kontraktų technologija grindžiamų sistemų kūrimo būdą, kuris gali palengvinti tokių sistemų kūrimo procesą.
2. Siūlomas MDA architektūra grindžiamas metodas palaiko reikalavimų specifikavimo, projektavimo etapus bei išmaniųjų kontraktų kodo generavimą, kurį galima panaudoti realizacijos etapo metu. Specifikuoti modeliai taip pat gali būti naudojami komunikacijos, dokumentacijos tikslais.
3. Siūlomas MDA architektūra grindžiamas metodas palaiko kelias platformas ir gali būti išplėstas skirtingoms išmaniųjų kontraktų technologijų platformoms palaikyti. Išplėtimas realizuojamas sukuriant papildomą *Blockchain PSM* UML profilį, modelių transformacijų tarp *PIM* ir *PSM* ir *PSM* - kodo transformacijų taisykles.

Rezultatų aprobavimas

Tyrimo rezultatai buvo paskelbti 5 moksliniuose leidiniuose: dvi publikacijos periodiniuose moksliniuose žurnaluose ir trys publikacijos konferencijų leidiniuose.

Disertacijos dokumento struktūra

Pirmajame skyriuje pateikiama blokų grandinės technologijų, išmaniųjų kontraktų kūrimo proceso ir modeliavimo taikymo programinės įrangos kūrimo

procese analizė. Taip pat pateikiama akademinės literatūros šaltiniuose rastų sprendimų lyginamoji analizė apie modeliais grindžiamus metodus, pritaikytus išmaniųjų kontraktų technologija grindžiamų sistemų ar išmaniųjų kontraktų kūrimo procesuose. Antrajame skyriuje pateikiamas MDA grindžiamas metodo pasiūlymas, išmaniųjų kontraktų technologija grindžiamoms sistemoms kurti. Trečiajame skyriuje pateikiamas eksperimentinis siūlomo metodo įvertinimas ir ketvirtajame pateikiamos išvados. Penktame skyriuje pateikiama disertacijos santrauka lietuvių kalba, šeštame skyriuje – literatūros šaltinių sąrašas, septintame – mokslinių leidinių ir konferencijų sąrašas ir galiausiai priedai.

1. EGZISTUOJANČIŲ SPRENDIMŲ IR METODŲ ANALIZĖ

Blokų grandinės apibrėžimas

Blokų grandinė yra paskirstyta duomenų bazė, kurioje saugomas nuolat augantis duomenų įrašų, apsaugotų nuo klastojimo ir peržiūros, sąrašas [2], *Satoshi Nakamoto* 2008 m. ją pristatė kaip kriptovaliutą, vadinamą *Bitcoin*. Iš esmės blokų grandinė yra decentralizuota, paskirstyta duomenų bazė, kuria dalijasi visi tinklo dalyviai [4]. Blokų grandinė nuo tradicinės duomenų bazės skiriasi duomenų saugojimo principais, kadangi bet kokio duomenų įrašymo į blokų grandinės viešąją knygą (angl. *public ledger*) atveju turi būti suformuota ir įvykdyta transakcija, jos duomenų patikrinimas ir pasiektas dalyvių konsensusas. Patikrinus ir išsaugojus transakcijos duomenis, šie pakeisti gali būti tik atliekant kitą operaciją, taigi transakcijų duomenys yra nekintami arba lengvai atsekami [5]. Taip blokų grandinės naudojimas skatina tinklo dalyvių tarpusavio pasitikėjimą, nes leidžia saugoti duomenis decentralizuotu, paskirstytu būdu, nepriklausomai nuo trečiųjų šalių [6]. Blokų grandinėje visi tinklo dalyviai turi viešosios knygos kopiją, o toks replikavimas leidžia atsekti duomenis ir jų pakeitimus. Duomenų paskirstymas tinkle užtikrina, kad viešoji knyga visada yra prieinama. Dėl to blokų grandinės transakcijų vykdymas iš esmės yra lėtesnis nei įprasta užklausa tradicinėje duomenų bazėje, nes kiekvienu atveju reikia suformuoti transakcijas, jas patvirtinti, jų pagrindu suformuoti bloką ir jį įrašyti į blokų grandinę.

Blokų grandinės technologija siekiama išspręsti keletą duomenų nuoseklumo, skaidrumo, vientisumo, prieinamumo ir patikimumo problemų [4] [10] [9] [11] [7]. Blokų grandinės taikymas siekia tai ištaisyti, padarant duomenis viešus ir bendrinamus visiems dalyviams, o bet kokie duomenų būsenos perėjimai yra vieši ir atliekami naudojant konsensuso algoritmus, siekiant užtikrinti, kad duomenų įterpimas būtų atliktas tik pasiekus tinklo dalyvių sutarimą [12].

Išmanusis kontraktas

Blokų grandinės technologijos palaiko ne tik kriptovaliutų, bet ir sudėtingesnių sprendimų realizaciją, tai galima pasiekti naudojant išmaniuosius kontraktus – programas, kurios yra saugomos ir vykdomos blokų grandinėje. Pagrindinis išmaniojo kontrakto tikslas yra automatiškai vykdyti sutarimo veiksmus, užfiksuotus programiniu kodu, kai įsigalioja tam tikros sąlygos. Kadangi išmanieji kontraktai yra diegiami į blokų grandinę, jie visais atvejais paveldi visas blokų grandinės

charakteristikas, todėl gali būti naudojami kaip įrankis įvairiems veiklos procesams decentralizuoti. Nors išmanieji kontraktai teoriškai galėtų apimti pilnus programinės įrangos taikomuosius sprendimus, dauguma tokių programų šiuo metu priklauso finansinei ar notarinei kategorijai.

Išmaniųjų kontraktų technologija grindžiama sistema

Blokų grandinė, naudojanti išmaniuosius kontraktus, leidžia realizuoti paskirstytą decentralizuotą programinę įrangą, kurios dalis arba visi komponentai yra talpinami lygiarangių architektūros (angl. *peer to peer, P2P*) tinkle ir kurios duomenys bendrinami naudojant viešąją knygą. Išmanieji kontraktai suteikia mechanizmą, leidžiantį apibrėžti ne tik susitarimus naudojant išmaniuosius kontraktus, bet ir visas programas, veikiančias blokų grandinės tinkle, nepasikliaujant tarpininkais. Dėl bet kokio duomenų įrašų įterpimo turi būti pasiektas dalyvių susitarimas, naudojant konsensuso algoritmus. Visi išmaniųjų kontraktų technologija grindžiamos sistemos komponentai, tokie kaip grafinė sąsaja [26], taip pat gali būti diegiami decentralizuotame tinkle, vis dėlto pagrindinis tokių sistemų komponentas yra išmanusis kontraktas, kuris visuomet diegiamas į blokų grandinę [33].

Išmaniųjų kontraktų technologija grindžiamų sistemų kūrimas

Informacinių sistemų srityje blokų grandinės technologijos taikymas vis dar yra gana ribotas [3]. Siekiant išplėsti taikymo sritis, ypatingas dėmesys turi būti skiriamas išmaniųjų kontraktų kūrimui, nes tai yra išmaniųjų kontraktų technologija grindžiamų sistemų pagrindas [26]. Šiuo metu išmaniųjų kontraktų kūrimas yra panašus į krioklio programinės įrangos kūrimo metodo modelį, nes įdiegtas išmanusis kontraktas negali būti atnaujintas, o kiti sistemos komponentai arba išmanusis kontraktas iki diegimo gali būti kuriami taikant iteratyvius kūrimo metodus [36] [13]. Dabartiniai išmaniųjų kontraktų ir blokų grandinės technologijomis grindžiamų sistemų kūrimo metodai yra besivystantys, vis dar reikalingos bendrosios kūrimo gairės, įrankiai, palengvinantys tokių sistemų kūrimą [26]. Šiuo metu išmaniųjų kontraktų kūrimo etapai nėra taip aiškiai apibrėžti, palyginti su tradiciniais programinės įrangos kūrimo metodais, yra kartais praleidžiami arba sujungiami su kitais kūrimo etapais ir neturi aiškiai apibrėžtų rezultatų. Kadangi kūrimo etapai nėra glaudžiai tarpusavyje susiję, tai verčia kūrėjus užpildyti kūrimo spragas patirtimi arba papildoma analize kiekviename etape [38]. Naudodamas šį labai specifinį, bet nepakankamai apibrėžtą ir sudėtingą kūrimo ciklą, kūrėjas gali praleisti svarbias detales, nes reikia susipažinti su pagrindiniais išmaniųjų kontraktų ir blokų grandinės technologija grindžiamų sprendimų kūrimo apribojimais [43]. Dėl to reikalingas bendras standartizuotas kūrimo metodas, kuris apimtų išmaniųjų kontraktų technologija grindžiamų sistemų struktūros ir elgsenos apibrėžimą, specifikaciją ir projektavimą ir galėtų palengvinti tokių sistemų kūrimo procesus.

Egzistuoja keletas mokslinių tyrimų krypčių, kuriose siūloma palengvinti išmaniųjų kontraktų technologija grindžiamų sistemų kūrimo procesą, automatizavimas yra viena iš jų. Juo siekiama palaikyti programinės įrangos

testavimo, diegimo ar kodo generavimo veiklas. Kadangi modeliais grindžiamų sistemų kūrimo metodai gali apimti įvairias veiklas ir būti naudojami reikalavimų specifikavimui, projektavimui ir realizacijai palaikyti [42], disertacijoje plačiau apžvelgiamos galimybės automatizuoti programinės įrangos kodų generavimo veiklas naudojant modeliais grindžiamus sistemų kūrimo metodus.

Modeliais grindžiamų sistemų kūrimo metodų taikymas kuriant išmaniųjų kontraktų technologija grindžiamas sistemas

Modeliais grindžiamų sistemų kūrimas yra programinės įrangos kūrimo metodas, kurio metu pagrindinis dėmesys skiriamas programinės įrangos koncepcinių modelių kūrimui [70]. Modeliai yra laikomi sistemų ar jų dalių abstrakcijomis ir naudojami kuriamoms sistemoms projektuoti ir analizuoti. Dalis modeliais grindžiamų kūrimo metodų papildomai naudoja modelius programinės įrangos artefaktų, kurie bus naudojami sistemos realizacijos ir diegimo veiklose, gamybai [45].

Atlikus lyginamąją modeliais grindžiamų sprendimų analizę, nustatyta, kad išmaniųjų kontraktų struktūrai specifikuoti dažnai naudojama UML klasių diagrama ir UML profiliai specifiniams blokų grandinės konceptams apibrėžti. Modeliais grindžiamo kūrimo metodai šių sistemų kūrimo proceso metu naudojami ne tik programinės įrangos realizacijos artefaktų gamyboje, bet modeliavimas palaikomas ir reikalavimų inžinerijos, projektavimo, modelių validavimo ir verifikavimo veiklose. Nors ir ne visi siūlymai yra susiję su automatiniu išmaniųjų kontraktų kodo generavimu, dauguma jų yra orientuoti į *Ethereum* platformos, konkrečiau *Solidity* programavimo kalbos, išmaniuosius kontraktus. Taip pat egzistuoja ir pasiūlymų, palaikančių kelias platformas, tuomet *Hyperledger* yra antroji dažnai pasirenkama blokų grandinės platforma.

Taip pat apžvelgus modeliais grindžiamų sprendimų taikymą dinaminio aspekto modeliavimo atveju nustatyta, kad modeliuojant išmaniųjų kontraktų elgseną labiausiai naudojamos yra BPMN ir UML notacijos. Notacijos pasirinkimas dalinai priklauso ir siūlomo metodo tikslų: jei pasiūlymai labiau pritaikyti veiklos procesams modeliuoti, yra naudojama BPMN, o jei metodai taikomi programinės įrangos elgsenos savybėms labiau specifikuoti, kartu naudojant apibrėžtas struktūrines detales, naudojama UML. Išmaniųjų kontraktų elgsenos modeliavimo metodų siūlymai dažnai palaiko išmaniųjų kontraktų programinio kodo generavimą iš elgsenos modelių, kuriuose naudojamos būsenų mašinos ar panašios notacijos diagramos. Vėlgi, apžvelgus modeliais grindžiamų sistemų kūrimo elgsenos modeliavimą taikančius metodus, pastebėta, kad dažniausiai pasirenkama platforma taip pat yra *Ethereum*, o antroji pagal pasirinkimą blokų grandinės technologija yra *Hyperledger Fabric* platforma.

MDA taikymas išmaniųjų kontraktų technologija grindžiamoms sistemoms kurti

Model Driven Architecture (toliau MDA) yra modeliais grindžiamų sistemų kūrimo metodas, skirtas naudoti modeliavimą reikalavimų, projektavimo, realizacijos, diegimo ir priežiūros veiklose. MDA teikia apibrėžtas gaires ir taikymo

principus, kuriais remiantis galima sukurti naujus modeliais grindžiamus metodus, kuriuose apibrėžiami keli abstrakcijos lygmenys. MDA aprašo tris skirtingų abstrakcijos lygmenų modelius – nuo veiklos modelio (*Computation Independent Model, toliau CIM*), nuo platformos nepriklausomo modelio (*Plaform Independent Model, toliau PIM*) ir platformai specializuoto modelio (*Platform Specific Model, toliau PSM*). Tradiciškai CIM skirtas konceptualiai apibūdinti sistemai ir jos kontekstui veiklos procesuose, PIM apibūdina sistemą be detalių apie konkrečią realizacijos technologijos platformą, o PSM apibūdina sistemą techninėmis detalėmis, būdingomis konkrečiai platformai, kurios yra būtinos programinės įrangos kodo generavimo metu. MDA pabrėžia modelių naudojimo svarbą modelis į modelį transformacijoms tarp skirtingų abstrakcijos lygmenų ir modelis į tekstą transformacijoms įgalinant kodo generavimą. MDA taip pat akcentuoja UML naudojimą kūrimo procese, kad būtų galima generuoti realizacijos artefaktus tiesiogiai iš programinės įrangos kūrimo proceso metu sudarytų modelių. MDA gali būti naudojamas kaip pagrindas siekiant apibrėžti įvairių sistemų, įskaitant išmaniųjų kontraktų technologijomis grindžiamų sistemų kūrimo procesus, atskiriant skirtingus programinės įrangos kūrimo proceso etapus. Tokio kūrimo metu modeliai gali būti transformuojami į žemesnio abstrakcijos lygio modelius ir šie modeliai galiausiai gali būti naudojami programinės įrangos realizacijos artefaktų kodui generuoti. Dauguma publikacijų apie kodo generavimą MDA grindžiamuose metoduose rodo, kad technologiją galima sėkmingai pritaikyti programinės įrangos kūrimui tobulinti [92] [47].

MDA grįsti išmaniųjų kontraktų technologijomis grindžiamų sistemų kūrimo metodai arba bent jau metodai, apibrėžiantys CIM, PIM ar PSM abstrakcijos lygmenis, yra išsamiau analizuojami disertacijoje [91] [89] [93] [71]. Deja, tik dalis metodų naudoja CIM abstrakcijos lygmenį arba jam prilyginamą modelį. Dalis siūlomų metodų išmaniųjų kontraktų kūrimo kontekste dažniausiai naudoja tekstines specifikacijas arba klasių diagramą. CIM yra abstrakcijos lygmuo, kuris, nors ignoruojamas net MDA architektūra grindžiamuose kūrimo metoduose, dažnai naudojamas veiklos procesų modeliams specifikuoti, taip pat CIM abstrakcija dažnai naudojama ir sistemos reikalavimams apibrėžti. Kai kuriais atvejais perėjimai nuo veiklos procesų iki sistemos reikalavimų pasiekiami naudojant modelių transformacijas, tačiau modelių transformacijos siūlomuose išmaniųjų kontraktų technologija grindžiamų sistemų kūrimo metoduose nėra taikomos. CIM abstrakcijos lygmens taikymas apžvelgtoje mokslinėje literatūroje ne tik neapibrėžtas, bet nėra ir vieningo požiūrio į CIM turinio apibrėžimą, tačiau galima pastebėti, kad dažniausiai naudojamos UML, BPMN notacijos ar tekstinės specifikacijos.

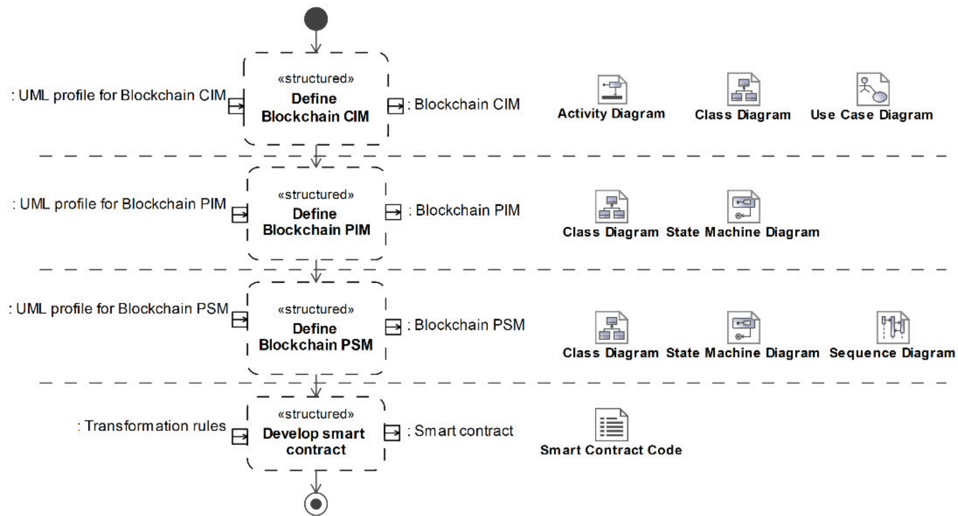
PIM abstrakcijos lygmuo apibrėžiamas daugumoje metodų, tačiau jis palaikomas labai įvairiais būdais, ir bendrų modeliavimo tendencijų būtent MDA architektūra grindžiamų metodų išmaniesiems kontraktams kurti nustatyti nepavyko. Analogiškai, kaip ir modeliais grindžiami sistemų kūrimo metodai, PIM abstrakcijos lygmeniu naudoja UML klasių diagramą išmaniųjų kontraktų struktūrai apibūdinti, o elgsenai modeliuoti naudoja būsenų mašinas ar panašias notacijas. MDA grindžiamų

metodų modelių transformacijoms realizuoti naudojama ATL kalba, tačiau labiausiai paplitęs metodas yra modelio transformacijas atlikti rankiniu būdu.

PSM abstrakcijos lygmeniu dažniausiai MDA grindžiamų metodų palaikoma platforma yra *Ethereum*. Siūlomi metodai daugiausia susiję su išmaniųjų kontraktų struktūros specifikacijomis, o būsenų mašinos naudojamos išmaniųjų kontraktų elgsenai modeliuoti. Dauguma apžvelgtų MDA grindžiamų metodų siūlymų yra skirti išmaniųjų kontraktų technologija grindžiamoms sistemoms kurti, tačiau automatinis kodo generavimas ar modelių transformacijos nėra plačiai palaikomos.

2. SIŪLOMAS MDA ARCHITEKTŪRA GRINDŽIAMAS METODAS IŠMANIŲJŲ KONTRAKTŲ TECHNOLOGIJA GRINDŽIAMOMS SISTEMOMS KURTI

Siekiant palengvinti ir automatizuoti išmaniųjų kontraktų technologija grindžiamų sistemų kūrimą, siūlomas MDA architektūros principais grįstas metodas. Pasiūlytas metodas leidžia modeliuoti išmaniųjų kontraktų technologija grindžiamų sistemų struktūrą ir elgseną naudojant UML modeliavimo kalbą. Metodas apima keturis išmaniųjų kontraktų technologija grindžiamų sistemų kūrimo etapus (2.1 pav.). Išmanusis kontraktas yra laikomas pagrindiniu išmaniųjų kontraktų technologija grindžiamos sistemos artefaktu [33], todėl siūlomo metodo kontekste pagrindinis dėmesys skiriamas būtent išmaniųjų kontraktų specifikacijai ir automatiniam šio realizacijos artefakto kodo generavimui. Kiekvienam siūlomo metodo modelio kūrimo etapui naudojamas UML profilis (2.1 pav., vaizduojama kaip įvestis), juo apibrėžiamas modelis, kuris vėliau naudojamas tolesnio etapo metu (2.1 pav., pateikiamas kaip išvestis). UML kartu su UML profiliais, skirtais *Blockchain CIM*, *Blockchain PIM* ir *Blockchain PSM*, yra naudojami išmaniųjų kontraktų technologija grindžiamų sistemų struktūrai ir elgsenai modeliuoti, o perėjimas tarp skirtingų abstrakcijos modelių yra automatizuotas naudojant modelių transformacijas. Galiausiai, sukurtas *Blockchain PSM* modelis yra panaudojamas generuojant išmaniojo kontrakto programinį kodą pasirinktai blokų grandinės platformai.



2.1 pav. MDA architektūra grindžiamas metodas išmaniųjų kontraktų technologija grindžiamoms sistemoms kurti

Pagal MDA architektūrą [117] [92], pirmasis kūrimo etapas yra *Blockchain CIM* apibrėžimas. Šio modelio tikslas pasiūlymo kontekste yra apibrėžti, kaip blokų grandinės technologija galėtų būti integruota į konkrečių veiklos procesų restruktūrizavimą, reorganizavimą ir decentralizavimą (modeliuojant UML veiklos diagramas), taip pat apibrėžti išmaniųjų kontraktų technologija grindžiamų sistemų reikalavimus kaip panaudojimo atvejų modelį (naudojant UML panaudojimo atvejų diagramą) ir dalykinės srities modelį (naudojant UML klasių diagramą). Panaudojimo atvejų modelis apibrėžia kuriamo sprendimo kontekstą, ko yra tikimasi iš sistemos, bei apibūdina integracijas su blokų grandinės technologija, tačiau paslepia visas su konkrečia blokų grandinės technologijos platforma susijusias detales [117] [92].

Tolesniame etape siūlomo metodo kontekste *Blockchain CIM* naudojamas kaip pagrindas *Blockchain PIM* kurti. Taikant *M2M* modelių transformacijas *Blockchain CIM* transformuojamas į *Blockchain PIM*. Vėliau šį modelį kūrėjas gali išplėsti, papildydamas išmaniojo kontrakto struktūrą arba specifikuodamas išmaniojo kontrakto elgseną. Tradiciškai PIM modelis specifikuoja sistemos projektą be išsamios informacijos apie jos realizaciją. Atsižvelgiant į tai, kad siūlomas metodas yra pritaikytas išmaniųjų kontraktų technologija grindžiamoms sistemoms kurti, pateiktame UML profilyje yra keletas konkrečių stereotipų, skirtų blokų grandinės technologijai būdingiems konceptams apibrėžti, tačiau jie yra pakankamai bendri, kad nebūtų paremti konkrečia blokų grandinės technologijos platforma.

Po *Blockchain PIM* sudarymo vyksta *Blockchain PSM* apibrėžimas. *Blockchain PSM* sudarymas taip pat atliekamas naudojant *M2M* modelių transformacijas ir anksčiau apibrėžtą *Blockchain PIM* modelį. Apibrėžtose *M2M* transformacijos taisyklėse nurodoma, kaip modeliai turi būti transformuojami į *Blockchain PSM* abstrakcijos lygmens modelį. Kadangi šio abstrakcijos lygmens

modeliai yra pritaikyti konkrečiai platformai, disertacijoje detalizuojamas dviejų platformų palaikymas, pritaikant PSM abstrakcijos lygmenį *Ethereum* ir *Hyperledger Fabric* blokų grandinės platformoms. Naudojant siūlomą metodą, galima modeliuoti *Solidity* ir *Go* programavimo kalbos išmaniųjų kontraktų struktūros ir elgsenos detales. Naudojant *M2T* modelių transformacijas, *Blockchain PSM* toliau naudojamas *Ethereum* arba (ir) *Hyperledger Fabric* platformų išmaniųjų kontraktų programiniam kodui sugeneruoti.

Šis metodas suteikia galimybę palengvinti ir iš dalies automatizuoti išmaniųjų kontraktų kūrimo procesą, pateikiant labiau struktūrizuotą požiūrį į išmaniųjų kontraktų technologijomis grindžiamų sistemų modeliavimą ir sudarant prielaidas išplėsti galimus blokų grandinės technologijos taikymus.

Kadangi metodas grindžiamas MDA architektūros principais, siūlomi MDA pagrįsti metodo kūrimo etapai – *Blockchain CIM*, *Blockchain PIM* ir dviejų *Blockchain PSM* (*Ethereum PSM* ir *Hyperledger Fabric PSM*) abstrakcijos lygmenys. Modelių turinys ir jų kūrimo gairės plačiau detalizuojami pagrindiniame disertacijos tekste.

3. EKSPERIMENTINIS TYRIMAS

Eksperimentiniai tyrimai buvo atlikti siekiant įvertinti siūlomo MDA architektūra grindžiamo metodo realizaciją dviem aspektais. Pirmojoje dalyje atliekamas išmaniųjų kontraktų programinio kodo artefaktų, sugeneruotų naudojant siūlomas modelių transformacijos taisykles, vertinimas. Antrojoje dalyje vertinamas siūlomo MDA architektūra grindžiamo metodo taikymas ir atliekamas sugeneruotų artefaktų įvertinimas išmaniųjų kontraktų technologija grindžiamų sistemų kūrimo proceso kontekste. Siūlomas išmaniųjų kontraktų technologija grindžiamų sistemų kūrimo metodas taip pat yra tiesiogiai palygintas su kitais MDA architektūra grindžiamais metodais.

Sugeneruotų išmaniųjų kontraktų kodo įvertinimas

Pagrindinis eksperimentinio įvertinimo tikslas yra pademonstruoti, kad siūlomo MDA architektūra grindžiamo metodo modelių transformacijos gali būti naudojamos *Solidity* ir *Go* išmaniųjų kontraktų elgsenai modeliuoti ir programiniam kodui sugeneruoti. Šiuo tikslu vertinami keli *Solidity* programavimo kalbos išmanieji kontraktai *Ethereum* platformai ir *Go* išmanusis kontraktas, skirtas *Hyperledger Fabric* platformai. Abiem atvejais išmanieji kontraktai buvo sumodeliuoti remiantis išmaniųjų kontraktų pavyzdžiais, pateiktais *Solidity* ir *Hyperledger Fabric* dokumentacijose [118] [119]. Galiausiai naudojant *M2T* modelių transformacijas sugeneruotas *Solidity* ir *Go* programavimo kalbos išmaniųjų kontraktų programinis kodas, kuris vėliau buvo įvertintas ir palygintas su originaliais dokumentacijoje pateikiamais išmaniųjų kontraktų pavyzdžiais pagal kodo ir vykdymo metrikas.

Remiantis pateiktais rezultatais galima teigti, kad siūlomas metodas gali automatizuoti *Ethereum* ir *Hyperledger Fabric* platformų išmaniųjų kontraktų programinio kodo generavimą, o sugeneruotų išmaniųjų kontraktų veikimas ir struktūra atitinka pateiktus dokumentacijose išmaniųjų kontraktų pavyzdžius.

Siūlomo MDA grindžiamo metodo taikymas

Siūlomas MDA grindžiamas metodas, skirtas išmaniųjų kontraktų technologija grindžiamoms sistemoms kurti, buvo panaudotas kuriant hakatono sertifikatų išdavimo sprendimą. Artefaktai, sukurti *Blockchain CIM*, *PIM* ir *PSM* specifikavimo metu, ir modelių transformacijos kiekybiniai rezultatai yra pateikiami disertacijos tekste. Be to, pateikiamas ir sugeneruotų išmaniųjų kontraktų programinio kodo metrikų vertinimas tiek *Solidity*, tiek *Go* programavimo kalbai, ir pateikiami transakcijų vykdymo rezultatai, gauti kontraktus įdiegus į testinius tinklus.

Pateikti rezultatai rodo, kad metodas gali būti taikomas išmaniųjų kontraktų kodo generavimui iš specifikuotų modelių automatizuoti; specifikuoti modeliai gali būti transformuoti į *Ethereum PSM* ir *Hyperledger Fabric PSM* modelius, taip patvirtinant, kad metodas palaiko kelias platformas.

MDA grindžiamų metodų palyginimas

Siūlomas išmaniųjų kontraktų technologija grindžiamų sistemos kūrimo metodas yra tiesiogiai palygintas su kitais MDA grindžiamais metodais. Palyginus su kitais MDA grindžiamais metodais, nustatyta, kad darbe siūlomas metodas vietoj tekstinių specifikacijų arba visai nenaudojamo CIM turi apibrėžtą CIM abstrakcijos lygmenį, kuris, naudojant UML, specifikuoja veiklos procesus, panaudojimo atvejų ir dalykinės srities modelius. Panašiai kaip ir kiti metodai, siūlomas metodas palaiko PIM ir PSM išmaniųjų kontraktų struktūros ir elgsenos modeliavimą naudojant būsenų mašinas, elgesio specifikacijai taip pat suteikiama alternatyva – sekų diagramos. Be to, siūlomas metodas įkomponuoja ir išmaniųjų kontraktų realizacijos standartus, kurie specifikuojami kaip PSM abstrakcijos lygio *opaque behavior* elementai. Galiausiai metodas palaiko kelias blokų grandinės technologijos platformas ir kodo generavimą į kelias programavimo kalbas.

4. IŠVADOS

1. Išmaniųjų kontraktų technologijomis grindžiamų sistemų kūrimo procesas yra labai specifinis kiekvienai blokų grandinės platformai, o preliminariuose kūrimo etapuose praleidžiama daug laiko nustatant technologijų ir konkrečių reikalavimų suderinamumą, ir tai, kad nėra nusistovėjusių specializuotų išmaniųjų kontraktų kūrimo metodų, apsunkina dabartinių išmaniųjų kontraktų technologijomis grindžiamų sistemų kūrimą. Analizė rodo, kad modeliavimas naudojamas programinės įrangos kūrimo veikloms palaikyti, nuo reikalavimų specifikavimo, sistemos projektavimo, validavimo, ar net kodo generavimui automatizuoti naudojant modelių transformacijas, dažnai propaguojamas *Model Driven Architecture* metoduose.
2. MDA architektūra grindžiamų metodų taikymo išmaniųjų kontraktų technologija grindžiamų sistemų artefaktams specifikuoti analizės metu pastebėta, kad dauguma pasiūlytų metodų yra koncepcinių stadijų, metodai nevisiškai išnaudoja skirtingus MDA abstrakcijos lygmenis: CIM abstrakcijos lygmuo dažnai ignoruojamas arba reikalavimai specifikuojami naudojant tekstines notacijas; PIM ir PSM lygmenys neturi aiškaus atskyrimo; PSM lygmuo kartais ignoruojamas ir kodas generuojamas tiesiogiai iš PIM. Dauguma

metodų orientuoti į *Solidity* programavimo kalbos išmaniųjų kontraktų kodo generavimą arba modelių validavimo veiklas, tik dalis siūlo daugiaplatformiškumą. Nė vienas iš analizuotų metodų nenaudoja modelio į modelį transformacijų, ir tik kai kurie naudoja modelis į tekstą transformacijas kodo generuoti.

3. Siūlomas MDA architektūra grindžiamas metodas, skirtas išmaniųjų kontraktų technologija grindžiamoms sistemoms kurti, apima *Blockchain CIM*, *Blockchain PIM* ir *Blockchain PSM* abstrakcijos lygmenis. Remiantis lyginamąja MDA architektūra grindžiamų metodų analize, siūlomas *Blockchain CIM* abstrakcijos lygis yra skirtas veiklos procesams ir išmaniųjų kontraktų technologija grindžiamos sistemos reikalavimams modeliuoti. *Blockchain PIM* abstrakcijos lygmuo naudojamas bendrinei išmaniojo kontrakto struktūrai ir elgsenai modeliuoti, nedetalizuojant konkrečios platformos principų, taip palaikant daugiaplatformiškumą. Aprašytas *Blockchain PSM* naudojamas konkrečios platformos išmaniojo kontrakto struktūrai ir elgsenai apibrėžti ir yra naudojamas transformacijose į programinį kodą. Panašiai kaip ir kiti MDA metodai, yra pritaikytas *Ethereum* ir *Hyperledger Fabric* platformoms, bet palaiko ir kitų išmaniųjų kontraktų realizacijos platformų išplėtimus.
4. Siūlomo MDA architektūra grindžiamo metodo *Blockchain CIM*, *PIM* ir *PSM* UML profiliai buvo realizuoti naudojant *Magicdraw* CASE įrankį, leidžiantį specifikuoti *Blockchain CIM, PIM, PSM* modelius, įgalinančius modelių transformacijas. *Blockchain CIM* naudoja *Blockchain CIM* UML profilį veiklos procesams ir išmaniųjų kontraktų technologija grindžiamos sistemos reikalavimams specifikuoti, integracijoms su blokų grandinės technologija, naudojant UML veiklos, panaudojimo atvejų ir klasių diagramas. Kaip ir kituose modeliais grindžiamuose kūrimo metoduose, išmaniųjų kontraktų struktūra yra modeliuojama UML klasių diagrama, naudojant realizuotą *Blockchain PIM* UML profilį, o elgsenos specifikacijai naudojama UML būsenų diagrama. *Blockchain PSM* abstrakcijos lygmuo naudojamas konkrečios blokų grandinės technologijos platformos išmaniojo kontrakto elgsenai ir struktūrai modeliuoti. Realizuotas *Blockchain PSM* UML profilis suteikia galimybę modeliuoti išmaniojo kontrakto struktūrą UML klasių diagrama, o elgsenai naudoti UML būsenų diagramas ir funkcijų elgsenai aprašyti UML sekų diagramas ar apibrėžti specifinių funkcijų elgseną naudojant *opaque behavior* specifikacijas.
5. Realizuotų modelių transformacijų derinys rodo, kad naudojant pasiūlytus MDA metodo UML plėtinius galima specifikuoti išmaniojo kontrakto struktūros ir elgsenos detales. Aukštesnio abstrakcijos lygio modelio informacija modelis į modelį transformacijos metu gali būti panaudojama žemesnio abstrakcijos lygio modelio detalėms apibrėžti. Taip pat modelis į tekstą transformacijos yra pritaikomos *Ethereum* platformos *Solidity* programavimo kalbos ir *Hyperledger Fabric* platformos *Go* programavimo kalbos išmaniųjų kontraktų kodui generuoti.
6. Įvertintos siūlomo MDA architektūra grindžiamo metodo išmaniųjų kontraktų technologija grindžiamoms sistemoms kurti modelių transformacijos, pademonstruojama, kad siūlomas metodas gali būti naudojamas išmaniojo

kontrakto struktūrai ir elgsenai modeliuoti naudojant UML klasių, būsenų mašinos ir sekų diagramas ir būti naudojamas vykdomo išmaniųjų kontraktų kodo generavimui automatizuoti. Sugeneruotas išmaniųjų kontraktų kodas iš modelių, parengtų remiantis *Solidity* ir *Hyperledger Fabric* dokumentacijoje aprašytais pavyzdžiais, įvertintas pagal kodo panašumą ir vykdymą. Vertinant kodo eilutes ir AST elementus, sugeneruotas ir originalus *Solidity* išmaniojo kontrakto kodas yra labai panašus, o apskaičiuoti panašumo matai svyruoja nuo 82% iki 99%; nustatyta, kad sugeneruotų *Go* kodo funkcijų ciklomatinis sudėtingumas yra identiškas originalui, taip pademonstruojant, kad transformacijos gali sugeneruoti išmaniųjų kontraktų kodą, panašų į originalą ir suteikiantį tą patį funkcionalumą.

7. MDA architektūra grindžiamas metodas išmaniųjų kontraktų technologija grindžiamoms sistemoms kurti buvo eksperimentiškai įvertintas taikant metodą hakatonų sertifikatų išdavimo sprendimo kūrimo metu. Rezultatai rodo, kad realizuotos metodo realizacija įgalina išmaniųjų kontraktų struktūros ir elgsenos specifikavimą, kas gali būti panaudota keliems skirtingiems specializuotos platformos modeliams sugeneruoti, o modelių transformacijos į vykdomąjį išmaniųjų kontraktų programinio kodą iš apibrėžtų *Ethereum PSM* ir *Hyperledger Fabric PSM* abstrakcijos lygmens modelių įgalina kelių platformų palaikymą.

REFERENCES

[1] A. R. Hevner, S. T. March, J. Park and S. Ram, "Design Science in Information Systems Research," *MIS Quarterly,* vol. 28, no. 1, pp. 75-105, 2004.

[2] S. Nakamoto, "Bitcoin: a peer-to-peer electronic cash system," 2018. [Online]. Available: https://bitcoin.org/bitcoin.pdf .

[3] F. Glaser, "Pervasive Decentralisation of Digital Infrastructures: A Framework for Blockchain enabled System and Use Case Analysis," in *Proceedings of the 50th Hawaii International Conference on System Sciences*, Hawaii, 2017.

[4] Y. Lu, "The blockchain: State-of-the-art and research challenges," *Journal of Industrial Information Integration,* vol. 15, pp. 80-90, 2019.

[5] M. Swan, Blockchain Blueprint for a New Economy, O'Reilly Media, 2015.

[6] S. Raval, Decentralized Applications: Harnessing Bitcoin's Blockchain Technology, O'Reilly Media, 2016.

[7] F. A. Sunny, P. Hajek, M. Munk, M. Z. Abedin, M. S. Satu, M. I. Alam and M. J. Islam, "A Systematic Review of Blockchain Applications," *IEEE Access,* vol. 10, pp. 59155-59177, 2022.

[8] Z. Zheng, S. Xie, H. Dai, X. Chen and H. Wang, "An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends," in *2017 IEEE International Congress on Big Data (BigData Congress)*, Honolulu, 2017.

[9] D. Berdik, S. Otoum, N. Schmidt, D. Porter and Y. Jararweh, "A Survey on Blockchain for Information Systems Management and Security," *Information Processing & Management,* vol. 58, no. 1, 2021.

[10] A. A. Monrat, O. Schelén and K. Andersson, "A Survey of Blockchain From the Perspectives of Applications, Challenges, and Opportunities," *IEEE Access,* vol. 7, pp. 117134-117151, 2019.

[11] P. Tasca and C. J. Tessone, "Taxonomy of Blockchain Technologies. Principles of Identication and Classication," *Ledger,* vol. 4, 2018.

[12] J. B. Bernabe, J. L. Canovas, J. L. Hernandez-Ramos, R. T. Moreno and A. Skarmeta, "Privacy-Preserving Solutions for Blockchain: Review and Challenges," *IEEE Access,* vol. 7, pp. 164908-164940, 2019.

[13] S. X. Zibin Zheng, H.-N. Dai, W. Chen, X. Chen, J. Weng and M. Imran, "An overview on smart contracts: Challenges, advances and platforms," *Future Generation Computer Systems,* vol. 105, pp. 475-491, 2020.

[14] F. Saleh, "Blockchain without Waste: Proof-of-Stake," *Review of Financial Studies,* vol. 34, no. 3, pp. 1156-1190, 2021.

[15] O. Ali, A. Jaradat, A. Kulakli and A. Abuhalimeh, "A Comparative Study: Blockchain Technology Utilization Benefits, Challenges and Functionalities," *IEEE Access,* vol. 9, pp. 12730-12749, 2021.

[16] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi and K.-L. Tan, "BLOCKBENCH: A Framework for Analyzing Private Blockchains," in *SIGMOD '17: Proceedings of the 2017 ACM International Conference on Management of Data*, Chicago, Illinois, USA , 2017.

[17] N. Darlington, "What is Blockchain Technology?," 2018. [Online]. Available:

https://blockgeeks.com/guides/what-is-blockchain-technology/.

[18] S. Omohundro, "Cryptocurrencies, smart contracts, and artificial intelligence," *AI Matters,* vol. 1, no. 2, pp. 19-21, 2014.

[19] M. Bartoletti and L. Pompianu, "An Empirical Analysis of Smart Contracts: Platforms, Applications, and Design Patterns," *arXiv: Cryptography and Security,* 2017.

[20] V. Buterin, "A Next-Generation Smart Contract and Decentralized Application," 2014. [Online]. Available: http://blockchainlab.com/pdf/Ethereum_white_paper-a_next_generation_smart_contract_and_decentralized_application_platform-vitalik-buterin.pdf.

[21] Q. Lu and X. Xu, "Adaptable Blockchain-Based Systems: A Case Study for Product Traceability," *IEEE Software,* vol. 34, no. 6, pp. 21-27, 2017.

[22] K. Wüst and A. Gervais, "Do you need a Blockchain?," in *2018 Crypto Valley Conference on Blockchain Technology (CVCBT)*, Zug, 2018.

[23] J. Yli-Huumo, D. Ko, S. Choi, S. Park and K. Smolander, "Where Is Current Research on Blockchain Technology?—A Systematic Review," *PLOS ONE,* vol. 11, no. 10, 2016.

[24] I. Mokdad and N. M. Hewahi, "Empirical Evaluation of Blockchain Smart Contracts," in *Decentralised Internet of Things*, Springer, 2020, p. 45–71.

[25] S. Dhaiouir and S. Assar, "A Systematic Literature Review of Blockchain-Enabled Smart Contracts: Platforms, Languages, Consensus, Applications and Choice Criteria," in *International Conference on Research Challenges in Information Science RCIS 2020*, 2020.

[26] W. Zou, D. Lo, P. S. Kochhar, X.-B. D. Le, X. Xia, Y. Feng, Z. Chen and B. Xu, "Smart Contract Development: Challenges and Opportunities," *IEEE Transactions on Software Engineering,* vol. 47, no. 10, pp. 2084-2106, 2019.

[27] G. Wood, "Ethereum: a Secure Decentralised Generalised Transaction Ledger," 2014. [Online]. Available: https://gavwood.com/paper.pdf.

[28] Hyperledger , "A Blockchain Platform for the Enterprise — hyperledger-fabricdocs main documentation," [Online]. Available: https://hyperledger-fabric.readthedocs.io.

[29] S. Wang, L. Ouyang, Y. Yuan, X. Ni, X. Han and F.-Y. Wang, "Blockchain-Enabled Smart Contracts: Architecture, Applications, and Future Trends," *IEEE Transactions on Systems, Man, and Cybernetics: Systems,* vol. 49, no. 11, pp. 2266-2277, 2019.

[30] "Ethereum Improvement Proposals," [Online]. Available: https://eips.ethereum.org/erc.

[31] OpenZeppelin, "OpenZeppelin Contracts," [Online]. Available: https://openzeppelin.com/contracts/.

[32] J. d. Kruijff and H. Weigand, "Understanding the Blockchain Using Enterprise Ontology," in *CAiSE: International Conference on Advanced Information Systems Engineering*, Essen, 2017.

[33] W. Metcalfe, "Ethereum, Smart Contracts, DApps," in *Blockchain and Crypto Currency*, Springer, 2020, pp. 77-93.

[34] C. Antal, T. Cioara, I. Anghel, M. Anta and I. Salomie, "Distributed Ledger Technology Review and Decentralized Applications Development Guidelines," *Future Internet,* vol. 13, no. 3, 2021.

[35] W. Cai, Z. Wang, J. B. Ernst, Z. Hong, C. Feng and V. C. M. Leung, "Decentralized

Applications: The Blockchain-Empowered Software System," *IEEE Access,* vol. 6, pp. 53019-53033, 2018.

[36] M. H. Miraz and M. Ali, "Blockchain Enabled Smart Contract Based Applications: Deficiencies with the Software Development Life Cycle Models," *Baltica Journal,* vol. 33, no. 1, pp. 101-116, 2020.

[37] A. Vacca, A. Di Sorbo, C. A. Visaggio and G. Canfora, "A systematic literature review of blockchain and smart contract development: Techniques, tools, and open challenges," *Journal of Systems and Software,* vol. 174, 2021.

[38] N. Sánchez-Gómez, J. Torres-Valderrama, J. A. García-García, J. J. Gutiérrez and M. J. Escalona, "Model-Based Software Design and Testing in Blockchain Smart Contracts: A Systematic Literature Review," *IEEE Access,* vol. 8, pp. 164556-164569, September 2020.

[39] S. Demi, M. Sánchez-Gordón and M. Kristiansen, "Blockchain for requirements traceability: A qualitative approach," *Journal of Software: Evolution and Process,* vol. e2493, 2022.

[40] M. Kondo, G. A. Oliva, Z. M. (. Jiang, A. E. Hassan and O. Mizuno, "Code cloning in smart contracts: a case study on verified contracts from the Ethereum blockchain platform," *Empirical Software Engineering,* vol. 25, pp. 4617-4675, 2020.

[41] Z. Gao, L. Jiang, X. Xia, D. Lo and J. Grundy, "Checking Smart Contracts with Structural Code Embedding," *IEEE Transactions on Software Engineering,* vol. 47, no. 12, pp. 2874-2891, 2021.

[42] Y. A. Hsain and N. L. S. Mbarki, "Ethereum's Smart Contracts Construction and Development using Model Driven Engineering Technologies: a Review," *Procedia Computer Science,* vol. 184, pp. 785-790, 2021.

[43] M. Fahmideh, J. Grundy, A. Ahmad, J. Shen, J. Yan, D. Mougouei, P. Wang, A. Ghose, A. Gunawardana, U. Aickelin and B. Abedin, "Engineering Blockchain Based Software Systems: Foundations, Survey, and Future Directions," *ACM Computing Surveys,* vol. 55, no. 6, pp. 1-44, 2022.

[44] J. Krogstie, "Modelling Languages: Perspectives and Abstraction Mechanisms," in *Model-Based Development and Evolution of Information Systems*, Springer, 2012.

[45] D. Akdur, V. Garousi and O. Demirörs, "A survey on modeling and model-driven engineering practices in the embedded software industry," *Journal of Systems Architecture,* vol. 91, pp. 62-82, 2018.

[46] A. L.´. Ramos, J.´. V. Ferreira and J. B.´o, "Model-Based Systems Engineering: An Emerging Approach for Modern Systems," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews),* vol. 42, no. 1, pp. 101-111, 2012.

[47] G. Sebastián, J. A.Gallud and R. Tesoriero, "Code generation using model driven architecture: A systematic mapping study," *Journal of Computer Languages,* vol. 56, 2020.

[48] Object Management Group, "Unified Modeling Language Specification Version 2.5," May 2015. [Online]. Available: https://www.omg.org/spec/UML/2.5.

[49] H. Koç, A. M. Erdoğan, Y. Barjakly and S. Peker, "UML Diagrams in Software Engineering Research: A Systematic Literature Review," *Proceedings,* vol. 74, no. 1, 2021.

[50] Object Management Group, "Business Process Model and Notation," [Online]. Available: http://www.bpmn.org.

[51] A. L. Campos and T. Oliveira, "Software Processes with BPMN: An Empirical Analysis," in *Product-Focused Software Process Improvement. PROFES 2013*, Paphos, 2013.

[52] Object Management Group, "Systems Modeling Language," [Online]. Available: http://www.omgsysml.org.

[53] S. Wolny, A. Mazak, C. Carpella, V. Geist and M. Wimmer, "Thirteen years of SysML: a systematic mapping study," *Software and Systems Modeling,* vol. 19, no. 1, 2020.

[54] J. J. Cadavid, B. Combemale and B. Baudry, "An analysis of metamodeling practices for MOF and OCL," *Computer Languages, Systems & Structures,* vol. 41, pp. 42-65, 2015.

[55] Object Management Group, "The Meta Object Facility Specification Version 2.5.1," October 2016. [Online]. Available: https://www.omg.org/spec/MOF/2.5.1.

[56] Object Management Group, "XML Metadata Interchange," [Online]. Available: https://www.omg.org/spec/XMI/2.5.1/PDF.

[57] L. Fuentes-Fernández and A. Vallecillo-Moreno, "An Introduction to UML Profiles," *UML and Model Engineering,* vol. 5, no. 2, 2004.

[58] T. Kosar, P. E. M. López, P. A. Barrientos and M. Mernika, "A preliminary study on various implementation approaches of domain-specific language," *Information and Software Technology,* vol. 50, no. 5, pp. 390-405, 2008.

[59] M. Rashid, M. W. Anwar and A. M. Khanc, "Toward the tools selection in model based system engineering for embedded systems—A systematic literature review," *Journal of Systems and Software,* vol. 106, pp. 150-163, 2015.

[60] Object Management Group, "The MOF Query/View/Transformation Specification Version 1.3," June 2016. [Online]. Available: https://www.omg.org/spec/QVT/1.3/PDF.

[61] The Eclipse Foundation, "ATL," [Online]. Available: https://www.eclipse.org/atl/.

[62] Object Management Group, "MOF Model to Text Transformation Language," January 2008. [Online]. Available: https://www.omg.org/spec/MOFM2T/1.0.

[63] R. Tesoriero, A. Rueda, J. A. Gallud, M. D. Lozano and A. Fernando, "Transformation Architecture for Multi-Layered WebApp Source Code Generation," *IEEE Access,* vol. 10, pp. 5223-5237, 2022.

[64] L. Huning, P. Iyenghar and E. Pulvermüller, "UML-based Model-Driven Code Generation of Error Detection Mechanisms," in *ICSEA 2020:The Fifteenth International Conference on Software Engineering Advances*, Porto, Portugal, 2020.

[65] S. Curty, F. Härer and H.-G. Fill, "Blockchain Application Development Using Model-Driven Engineering and Low-Code Platforms: A Survey," in *International Conference on Business Process Modeling, Development and Support, International Conference on Evaluation and Modeling Methods for Systems Analysis and Development*, Leuven, 2022.

[66] L. Marchesi, M. Marchesi and R. Tonelli, "ABCDE—agile block chain DApp engineering," *Blockchain: Research and Applications,* vol. 1, no. 1-2, 2020.

[67] H. Rocha and S. Ducasse, "Preliminary Steps Towards Modeling Blockchain Oriented Software," in *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, Gothenburg, 2018.

[68] H. M. Kim and M. Laskowski, "Toward an ontology-driven blockchain design for supply-chain provenance," *Intelligent Systems in Accounting, Finance and Management,* vol. 25, no. 1, pp. 18-27, 2018.

[69] G. A. Pierro, "Smart-Graph: Graphical Representations for Smart Contract on the Ethereum Blockchain," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Honolulu, 2021.

[70] S. Kent, "Model Driven Engineering," 2002. [Online]. Available: https://link.springer.com/chapter/10.1007/3-540-47884-1_16.

[71] K. Hu, J. Zhu, Y. Ding, X. Bai and J. Huang, "Smart Contract Engineering," *Electronics,* vol. 9, no. 12, 2020.

[72] F. Ciccozzi, I. Malavolta and B. Selic, "Execution of UML models: a systematic review of research and practice," *Software and Systems Modeling,* vol. 18, p. 2313–2360 , 2019.

[73] M. Hamdaqa, L. A. P. Metz and I. Qasse, "iContractML: A Domain-Specific Language for Modeling and Deploying Smart Contracts onto Multiple Blockchain Platforms," in *SAM '20: Proceedings of the 12th System Analysis and Modelling Conference*, 2020.

[74] M. Hamdaqa, L. A. Pined, Met and I. Qasse, "iContractML 2.0: A domain-specific language for modeling and deploying smart contracts onto multiple blockchain platforms," *Information and Software Technology,* vol. 144, 2022.

[75] M. Skotnica, J. Klicpera and R. Pergl, "Towards Model-Driven Smart Contract Systems - Code Generation and Improving Expressivity of Smart Contract Modeling," in *Proceedings of the 20th CIAO! Doctoral Consortium, and Enterprise Engineering Working Conference Forum 2020*, Bolzano, 2020.

[76] M. Skotnica and R. Pergl, "Das Contract - A Visual Domain Specific Language for Modeling Blockchain Smart Contracts," in *Enterprise Engineering Working Conference*, Lisbon, 2020.

[77] T. Górski and J. Bednarski, "Applying Model-Driven Engineering to Distributed Ledger Deployment," *IEEE Access,* vol. 8, pp. 118245-118261, 2020.

[78] T. Górski and J. Bednarski, "Transformation of the UML Deployment Model into a Distributed Ledger Network Configuration," in *2020 IEEE 15th International Conference of System of Systems Engineering (SoSE)*, Budapest, 2020.

[79] T. Górski, "Continuous Delivery of Blockchain Distributed Applications," *Sensors,* vol. 22, no. 1, 2022.

[80] P. Garamvölgyi, I. Kocsis, B. Gehl and A. 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)*, Luxembourg, 2018.

[81] A. Mavridou, A. Laszka, E. Stachtiari and A. Dubey, "VeriSolid: Correct-by-Design Smart Contracts for Ethereum," in *FC: International Conference on Financial Cryptography and Data Security*, Frigate Bay, 2019.

[82] D. Suvorov and V. Ulyantsev, "Smart Contract Design Meets State Machine Synthesis: Case Studies," 2019.

[83] V. Ulyantsev, I. Buzhinsky and A. Shalyto, "Exact finite-state machine identification from scenarios and temporal properties," *International Journal on Software Tools for Technology Transfer,* vol. 20, pp. 35-55, 2018.

[84] N. Zupan, P. Kasinathan, J. Cuellar and M. Sauer, "Secure Smart Contract Generation Based on Petri Nets," in *Blockchain Technology for Industry 4.0. Blockchain Technologies*, Springer, 2020.

[85] P. Kasinathan, D. Martintoni, B. Hofmann, V. Senni and M. Wimmer, "Secure Remote Maintenance via Workflow-Driven Security Framework," in *2021 IEEE International Conference on Blockchain (Blockchain)*, Melbourne, 2021.

[86] O. Choudhury, N. Rudolph, I. Sylla, N. Fairoza and A. Das, "Auto-Generation of Smart Contracts from Domain-Specific Ontologies and Semantic Rules," in *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, Halifax, 2018.

[87] Q. Lu, A. B. Tran, I. Weber, H. O'Connor, P. Rimba, X. Xu, M. Staples, L. Zhu and R. Jeffery, "Integrated model-driven engineering of blockchain applications for business processes and asset management," *Software: Practice and Experience,* vol. 51, no. 5, pp. 1059-1079, 2021.

[88] O. López-Pintado, L. García-Bañuelos, M. Dumas, I. Weber and A. Ponomarev, "Caterpillar: A business process execution engine on the Ethereum blockchain," *Software: Practice and Experience,* vol. 49, no. 7, pp. 1162-1193, 2019.

[89] H. Syahputra and H. Weigand, "The Development of Smart Contracts for Heterogeneous Blockchains," in *Proceedings of the I-ESA Conferences*, 2019.

[90] J. d. Kruijff and H. Weigand, "Towards a Blockchain Ontology," 2017.

[91] K. Boogaard, *A Model-Driven Approach to Smart Contract Development,* Faculty of Science Theses, 2018.

[92] Object Management Group, "Model Driven Architecture (MDA) MDA Guide rev. 2.0," 18 June 2014. [Online]. Available: https://www.omg.org/cgi-bin/doc?ormsc/14-06-01.

[93] C. B. Victor Amaral de Sousa, "MDE4BBIS: A Framework to Incorporate Model-Driven Engineering in the Development of Blockchain-Based Information Systems," in *IEEE International Conference on Blockchain Computing and Applications*, Tartu, 2021.

[94] V. A. d. Sousa, C. Burnay and M. Snoeck, "B-MERODE: A Model-Driven Engineering and Artifact-Centric Approach to Generate Blockchain-Based Information Systems," in *CAiSE: International Conference on Advanced Information Systems Engineering*, Grenoble, 2020.

[95] M. Drozdova, M. Kardos, Z. Kurillova and B. Bucko, "Transformation in Model Driven Architecture," in *Information Systems Architecture and Technology: Proceedings of 36th International Conference on Information Systems Architecture and Technology – ISAT 2015* , 2016.

[96] Y. Rhazali, Y. Hadi and A. Mouloudi, "Disciplined Approach for Transformation CIM to PIM in MDA," in *3rd International Conference on Model-Driven Engineering and Software*, Angers, 2015.

[97] A. Rodríguez, I. G.-R. d. Guzmán, E. Fernández-Medina and M. Piattini, "Semi-formal transformation of secure business processes into analysis class and use case models: An MDA approach," *Information and Software Technology,* vol. 52, no. 9, pp. 945-971, 2010.

[98] O. E. Beggar, K. Letrache and M. Ramdani, "CIM for data warehouse requirements

using an UML profile," *IET Software,* vol. 11, no. 4, pp. 181-194, 2017.

[99] M. Melouk, Y. Rhazalib and H. Youssef, "An Approach for Transforming CIM to PIM up To PSM in MDA," *Procedia Computer Science,* vol. 170, pp. 869-874, 2020.

[100] N. Prat, J. Akoka and I. Comyn-Wattiau, "An MDA approach to knowledge engineering," *Expert Systems with Applications,* vol. 39, no. 12, pp. 10420-10437, 2012.

[101] M. F. Amr, N. Benmoussa, K. Mansouri and M. Qbadou, "Transformation of the CIM Model into A PIM Model According to The MDA Approach for Application Interoperability: Case of the "COVID-19 Patient Management" Business Process," *International Journal of Online and Biomedical Engineering ,* vol. 17, no. 5, pp. 49-68, 2021.

[102] J. Osis, E. Asnina and A. Grave, "Computation Independent Modeling within the MDA," in *IEEE International Conference on Software-Science*, Herzlia, 2007.

[103] I. Essebaa and C. Salima, "QVT Transformation Rules to Get PIM Model from CIM Model," in *Europe and MENA Cooperation Advances in Information and Communication Technologies*, Saidia, 2016.

[104] I. Essebaa and S. Chantit, "Toward an automatic approach to get PIM level from CIM level using QVT rules," in *2016 11th International Conference on Intelligent Systems: Theories and Applications (SITA)*, Mohammedia, 2016.

[105] I. Essebaa, S. Chantit and M. Ramdani, "MoDAr-WA: Tool Support to Automate an MDA Approach for MVC Web Application," *Computers,* vol. 8, no. 4, 2019.

[106] N. Kharmoum, S. Ziti, Y. Rhazali and F. Omary, "AN AUTOMATIC TRANSFORMATION METHOD FROM THE E3VALUE MODEL TO UML2 SEQUENCE DIAGRAMS: AN MDA APPROACH," *International Journal of Computing,* vol. 18, no. 3, pp. 316-330, 2019.

[107] K. Letrache, O. E. Beggar and M. Ramdani, "The automatic creation of OLAP cube using an MDA approach," *Software: Practice and Experience,* vol. 47, no. 12, pp. 1887-1903, 2017.

[108] J. Osis and E. Asnina, "Enterprise Modeling for Information System Development within MDA," in *Proceedings of the 41st Annual Hawaii International Conference on System Sciences*, Waikoloa, 2008.

[109] M. Kardos and M. Drozdova, "Analytical method of CIM to PIM transformation in Model Driven Architecture (MDA)," *Journal of Information and Organizational Sciences,* vol. 34, no. 1, 2010.

[110] R. Paulavičius, S. Grigaitis and E. Filatovas, "A Systematic Review and Empirical Analysis of Blockchain Simulators," *IEEE Access,* vol. 9, pp. 38010-38028, 2021.

[111] E. Filatovas, M. Marcozzi, L. Mostarda and R. Paulavičiusa, "A MCDM-based framework for blockchain consensus protocol selection," *Expert Systems with Applications,* vol. 204, 2022.

[112] P. Danielius, P. Stolarski and S. Masteika, "Vulnerabilities and Excess Gas Consumption Analysis Within Ethereum-Based Smart Contracts for Electricity Market," in *Business Information Systems Workshops. BIS 2020*, Colorado Springs, 2020.

[113] A. Skaržauskienė, M. Mačiulienė and D. Bar, "Developing Blockchain Supported Collective Intelligence in Decentralized Autonomous Organizations," in *Proceedings of the Future Technologies Conference (FTC) 2020*, Vancouver, 2020.

[114] T. Skersys, P. Danėnas, R. Butleris, A. Ostreika and J. Čeponis, "Extracting SBVR Business Vocabularies from UML Use Case Models Using M2M Transformations Based on Drag-and-Drop Actions," *Applied Sciences,* vol. 11, no. 14, 2021.

[115] S. Gudas and A. Valatavičius, "Extending Model-Driven Development Process with Causal Modeling Approach," in *Data Science: New Issues, Challenges and Applications*, Springer, 2020.

[116] I. Veitaitė and A. Lopata, "Knowledge-Based Transformation Algorithms of UML Dynamic Models Generation from Enterprise Model," in *Data Science: New Issues, Challenges and Applications*, Springer, 2020.

[117] O. Pastor and J. C. Molina, Model-Driven Architecture in Practice, Springer, 2007.

[118] Ethereum, "Solidity documentation," [Online]. Available: https://docs.soliditylang.org/.

[119] Hyperledger, "Hyperledger-fabricdocs main documentation," [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-2.2/write_first_app.html.

[120] Hyperledger, "Hyperledger Fabric Samples," [Online]. Available: https://github.com/hyperledger/fabric-samples.

[121] M. Jurgelaitis, L. Čeponienė and R. Butkienė, "Solidity Code Generation From UML State Machines in Model-Driven Smart Contract Development," *IEEE Access,* vol. 10, pp. 33465-33481, 2022.

[122] ConsenSys Software, "VSCode Solidity Metrics," [Online]. Available: https://github.com/ConsenSys/vscode-solidity-metrics/.

[123] Remix Project, "Remix - Ethereum IDE," [Online]. Available: https://remix.ethereum.org/.

[124] Ethereum, "Solidity by Example | Simple Open Auction," [Online]. Available: https://docs.soliditylang.org/en/v0.8.0/solidity-by-example.html#simple-open-auction.

[125] Ethereum, "Solidity by Example | Safe Remote Purchase," [Online]. Available: https://docs.soliditylang.org/en/v0.8.0/solidity-by-example.html#safe-remote-purchase.

[126] "Common Patterns | State Machine," Ethereum , [Online]. Available: https://docs.soliditylang.org/en/v0.8.0/common-patterns.html#state-machine.

[127] M. Jurgelaitis, L. Čeponienė, K. Butkus, R. Butkienė and V. Drungilas, "MDA-Based Approach for Blockchain Smart Contract Development," *Applied Sciences,* vol. 13, no. 1, 2022.

[128] J. Feist, G. Grieco and A. Groce, "Slither: A Static Analysis Framework for Smart Contracts," in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, Montréal, 2019.

[129] P. Li, S. Li, M. Ding, J. Yu, H. Zhang, X. Zhou and J. Li, "A Vulnerability Detection Framework for Hyperledger Fabric Smart Contracts Based on Dynamic and Static Analysis," in *EASE '22: Proceedings of the International Conference on Evaluation and Assessment in Software Engineering 2022*, Gothenburg, 2022.

SCIENTIFIC PUBLICATION AND CONFERENCE LIST

Scientific publications in Periodicals

1. **Jurgelaitis, Mantas**; Čeponienė, Lina; Butkus, Karolis; Butkienė, Rita; Drungilas, Vaidotas. MDA-based approach for blockchain smart contract development // Applied sciences. Basel : MDPI. ISSN 2076-3417. 2023, vol. 13, iss. 1, art. No. 487, p. 1–28. DOI: 10.3390/app13010487.
2. **Jurgelaitis, Mantas**; Čeponienė, Lina; Butkienė, Rita. Solidity Code Generation from UML State Machines in Model-Driven Smart Contract Development // IEEE Access. Piscataway, NJ: IEEE. ISSN 2169-3536. 2022, vol. 10, p. 33465–33481. DOI: 10.1109/ACCESS.2022.3162227.

Scientific publications in Conference Proceedings

1. **Jurgelaitis, M.**; Čeponienė, L.; Butkienė, R.; Valatkevičius, T. Computation Independent Model in MDA-based Smart Contract Development // Proceedings of the 1st blockchain and cryptocurrency conference (B2C' 2022), 9-11 November 2022, Barcelona, Spain / edited by Sergey Y. Yurish. Barcelona : IFSA publishing, 2022. eISBN 9788409457632. p. 20–22. Available online: https://sensorsportal.com/B2C/B2C_2022_Proceedings.pdf
2. **Jurgelaitis, Mantas**; Drungilas, Vaidotas; Čeponienė, Lina; Vaičiukynas, Evaldas; Butkienė, Rita; Čeponis, Jonas. Smart Contract Code Generation from Platform Specific Model for Hyperledger Go // Trends and applications in information systems and technologies: World conference on information systems and technologies, WorldCIST' 21, Angra do Heroismo city, Terceira Island, Azores, Portugal, 30 March – 2 April 2021 / Rocha Á., Adeli H., Dzemyda G., Moreira F., Ramalho Correia A.M. (eds.). Cham : Springer, 2021. ISBN 9783030726539. eISBN 9783030726546. p. 63-73. (Advances in intelligent systems and computing, ISSN 2194-5357, eISSN 2194-5365 ; vol. 1368). DOI: 10.1007/978-3-030-72654-6_7.
3. **Jurgelaitis, Mantas**; Butkienė, Rita; Vaičiukynas, Evaldas; Drungilas, Vaidotas; Čeponienė, Lina. Modelling Principles for Blockchain-based Implementation of Business or Scientific Processes // CEUR workshop proceedings: IVUS 2019 international conference on information technologies: proceedings of the international conference on information technologies, Kaunas, Lithuania, April 25, 2019 / edited by: Robertas Damaševičius, Tomas Krilavičius, Audrius Lopata, Dawid Połap, Marcin Woźniak. Aachen : CEUR-WS. ISSN 1613-0073. 2019, vol. 2470, p. 43–47. Available Online: http://ceur-ws.org/Vol-2470/p13.pdf

APPENDIXES

Appendix 1. Blockchain CIM to PIM Model transformation rules

Transformation from CIM Use Cases to PIM Operations

```
rule UseCase {
    from source : UML!"uml::UseCase" (
                 source.subject.notEmpty() and source.subject.oclIsKindOf(UML!"uml::Actor")
and source.subject.hasStereotype('blockchain')
                                  )
    to t : UML!"uml::Operation" (
        name <- source.name,
        visibility <- source.visibility,
        isLeaf <- source.isLeaf,
        isAbstract <- source.isAbstract,
        eAnnotations <- source.eAnnotations,
        ownedComment <- source.ownedComment,
        ownedRule <- source.ownedRule,
        ownedBehavior <- source.ownedBehavior,
        subject <- source.subject)
}
```

Transformation from CIM Classes to PIM Contained Classes

```
rule ERtoContainedClass {
    from
        source: UML!"uml::Class" in IN (
            source.hasStereotype('on-chain')
        )
    to
        t: UML!"uml::Class" (
            name <- source.name,
            visibility <- source.visibility,
            isLeaf <- source.isLeaf,
            isAbstract <- source.isAbstract,
            isActive <- source.isActive,
            eAnnotations <- source.eAnnotations,
            ownedComment <- source.ownedComment,
            nameExpression <- source.nameExpression,
            elementImport <- source.elementImport,
            packageImport <- source.packageImport,
            ownedRule <- source.ownedRule,
            templateParameter <- source.templateParameter,
            templateBinding <- source.templateBinding,
            ownedTemplateSignature <- source.ownedTemplateSignature,
            generalization <- source.generalization,
            powertypeExtent <- source.powertypeExtent,
            redefinedClassifier <- source.redefinedClassifier,
            substitution <- source.substitution,
            representation <- source.representation,
            collaborationUse <- source.collaborationUse,
            ownedUseCase <- source.ownedUseCase,
            useCase <- source.useCase,
            ownedAttribute <- source.ownedAttribute,
            ownedConnector <- source.ownedConnector,
            ownedBehavior <- source.ownedBehavior,
             classifierBehavior <- source.classifierBehavior,
            interfaceRealization <- source.interfaceRealization,
            nestedClassifier <- source.nestedClassifier,
            ownedOperation <- source.ownedOperation,
            ownedReception <- source.ownedReception
        )
}
```

Transformation from CIM Properties to PIM Contained Class Properties

```
rule ERtoContainedClass {
    from
        source: UML!"uml::Class" in IN (
            source.hasStereotype('on-chain')
        )
    to
        target: UML!"uml::Class" (
            name <- source.name,
            visibility <- source.visibility,
            isLeaf <- source.isLeaf,
            isAbstract <- source.isAbstract,
            isActive <- source.isActive,
            eAnnotations <- source.eAnnotations,
            ownedComment <- source.ownedComment,
            nameExpression <- source.nameExpression,
            elementImport <- source.elementImport,
            packageImport <- source.packageImport,
            ownedRule <- source.ownedRule,
            templateParameter <- source.templateParameter,
            templateBinding <- source.templateBinding,
            ownedTemplateSignature <- source.ownedTemplateSignature,
            generalization <- source.generalization,
            powertypeExtent <- source.powertypeExtent,
            redefinedClassifier <- source.redefinedClassifier,
            substitution <- source.substitution,
            representation <- source.representation,
            collaborationUse <- source.collaborationUse,
            ownedUseCase <- source.ownedUseCase,
            useCase <- source.useCase,
            ownedAttribute <-
            if(source.ownedAttribute.hasStereotype('on-chain'))
                then source.ownedAttribute
                else source.destroy()
            endif,
            ownedConnector <- source.ownedConnector,
            ownedBehavior <- source.ownedBehavior,
            classifierBehavior <- source.classifierBehavior,
            interfaceRealization <- source.interfaceRealization,
            nestedClassifier <- source.nestedClassifier,
            ownedOperation <- source.ownedOperation,
            ownedReception <- source.ownedReception
        )
}
```

Transformation from CIM Member End to PIM Contained Class Properties

```
rule Association {
    from source : UML!"uml::Association" in IN (source.oclIsTypeOf(UML!"uml::Association"))
    to t : UML!"uml::Association" (
        name <- source.name,
        visibility <- source.visibility,
        isLeaf <- source.isLeaf,
        isAbstract <- source.isAbstract,
        isDerived <- source.isDerived,
        eAnnotations <- source.eAnnotations,
        ownedComment <- source.ownedComment,
        clientDependency <- source.clientDependency,
        nameExpression <- source.nameExpression,
        elementImport <- source.elementImport,
        packageImport <- source.packageImport,
        ownedRule <- source.ownedRule,
        templateParameter <- source.templateParameter,
        templateBinding <- source.templateBinding,
        ownedTemplateSignature <- source.ownedTemplateSignature,
        generalization <- source.generalization,
        powertypeExtent <- source.powertypeExtent,
        redefinedClassifier <- source.redefinedClassifier,
        substitution <- source.substitution,
        representation <- source.representation,
        collaborationUse <- source.collaborationUse,
        ownedUseCase <- source.ownedUseCase,
        useCase <- source.useCase,
        memberEnd <-
       if(source.memberEnd.hasStereotype('on-chain'))
           then source.memberEnd
           else source.destroy()
       endif,
       navigableOwnedEnd <- source.navigableOwnedEnd)
}
```

Appendix 2. Blockchain PIM to Ethereum PSM transformation rules

Transformation from PIM SmartContract to Ethereum PSM Contract

```
rule SmartContract2Contract {
    from
    source: UML!"uml::Class" in IN (
        source.oclIsTypeOf(UML!"uml::Class")
    )
     using {
        target : UML!"uml::Class" = source.resolve();
    }
    to
    t: UML!"uml::Class" (
        name <- source.name,
        visibility <- source.visibility,
        isLeaf <- source.isLeaf,
        isAbstract <- source.isAbstract,
        isActive <- source.isActive,
        eAnnotations <- source.eAnnotations,
        ownedComment <- source.ownedComment,
        nameExpression <- source.nameExpression,
        elementImport <- source.elementImport,
        packageImport <- source.packageImport,
        ownedRule <- source.ownedRule,
        templateParameter <- source.templateParameter,
        templateBinding <- source.templateBinding,
        ownedTemplateSignature <- source.ownedTemplateSignature,
        generalization <- source.generalization,
        powertypeExtent <- source.powertypeExtent,
        redefinedClassifier <- source.redefinedClassifier,
        substitution <- source.substitution,
        representation <- source.representation,
        collaborationUse <- source.collaborationUse,
        ownedUseCase <- source.ownedUseCase,
        useCase <- source.useCase,
        ownedAttribute <- source.ownedAttribute,
        ownedConnector <- source.ownedConnector,
         ownedBehavior <- source.ownedBehavior,
            classifierBehavior <- source.classifierBehavior,
        interfaceRealization <- source.interfaceRealization,
        nestedClassifier <- source.nestedClassifier,
        ownedOperation <- source.ownedOperation,
        ownedReception <- source.ownedReception
    )
    do{
        if(source.hasStereotype('SmartContract')) {
            for(as in source.getApplicableStereotypes()){
                if(as.name = 'contract' and target.hasStereotype('contract') = false){
                    target.applyStereotype(as);
                    for (a in as.getAllAttributes()) {
                        if (not a.name.startsWith('base_Class') and source.hasValue(as, a.name))
{
                            target.setValue(as, a.name, source.getValue(as, a.name));
                        }
                    }
                }
            }
        }
    }
}
```

Transformation from PIM Contained Classes to Ethereum PSM Struct Class

```
rule ContainedClass2Struct {
    from
    source: UML!"uml::Class" in IN (
        source.oclIsTypeOf(UML!"uml::Class")
    )
    using {
        target : UML!"uml::Class" = source.resolve();
    }
    to
    t: UML!"uml::Class" (
        name <- source.name,
        visibility <- source.visibility,
        isLeaf <- source.isLeaf,
        isAbstract <- source.isAbstract,
        isActive <- source.isActive,
        eAnnotations <- source.eAnnotations,
        ownedComment <- source.ownedComment,
        nameExpression <- source.nameExpression,
        elementImport <- source.elementImport,
        packageImport <- source.packageImport,
        ownedRule <- source.ownedRule,
        templateParameter <- source.templateParameter,
        templateBinding <- source.templateBinding,
        ownedTemplateSignature <- source.ownedTemplateSignature,
        generalization <- source.generalization,
        powertypeExtent <- source.powertypeExtent,
        redefinedClassifier <- source.redefinedClassifier,
        substitution <- source.substitution,
        representation <- source.representation,
        collaborationUse <- source.collaborationUse,
        ownedUseCase <- source.ownedUseCase,
        useCase <- source.useCase,
        ownedAttribute <- source.ownedAttribute,
        ownedConnector <- source.ownedConnector,
        ownedBehavior <- source.ownedBehavior,
            classifierBehavior <- source.classifierBehavior,
        interfaceRealization <- source.interfaceRealization,
        nestedClassifier <- source.nestedClassifier,
        ownedOperation <- source.ownedOperation,
        ownedReception <- source.ownedReception
    )
    do{
        if(source.owner.hasStereotype('SmartContract')) {
            for(as in source.getApplicableStereotypes()){
                if(as.name = 'struct' and target.hasStereotype('struct') = false){
                    target.applyStereotype(as);
                    for (a in as.getAllAttributes()) {
                        if (not a.name.startsWith('base_Class') and source.hasValue(as, a.name))
{
                            target.setValue(as, a.name, source.getValue(as, a.name));
                        }
                    }
                }
            }
        }
    }
}
```

Transformation from PIM Contained Class Properties to Ethereum PSM Struct Class Member Properties

```
rule Property2Member {
    from
        source: UML!"uml::Property" in IN
    using {
        target : UML!"uml::Property" = source.resolve();
    }
    to
        t: UML!"uml::Property" (
            aggregation <- source.aggregation,
            association <- source.association,
            defaultValue <- source.defaultValue,
            deployment <- source.deployment,
            eAnnotations <- source.eAnnotations,
            isDerived <- source.isDerived,
            isDerivedUnion <- source.isDerivedUnion,
            isLeaf <- source.isLeaf,
            isOrdered <- source.isOrdered,
            isReadOnly <- source.isReadOnly,
            isStatic <- source.isStatic,
            isUnique <- source.isUnique,
            lowerValue <- source.lowerValue,
            name <- source.name,
            nameExpression <- source.nameExpression,
            ownedComment <- source.ownedComment,
            qualifier <- source.qualifier,
            redefinedProperty <- source.redefinedProperty,
            subsettedProperty <- source.subsettedProperty,
            templateParameter <- source.templateParameter,
            type <- source.type,
            upperValue <- source.upperValue,
            visibility <- source.visibility
        )
        do{
            if(source.isStereotypeApplied().oclIsUndefined()) {
                for(as in source.getApplicableStereotypes()){
                    if(as.name = 'member' and target.hasStereotype('member') = false){
                        target.applyStereotype(as);
                    }
                }
            }
        }
}
```

Transformation from PIM Enumeration to Ethereum PSM Enum Enumeration

```
rule Enumeration2Enum {
    from source : UML!"uml::Enumeration" in IN
    using {
        target : UML!"uml::Enumeration" = source.resolve();
    }
    to t : UML!"uml::Enumeration" (
        name <- source.name,
        visibility <- source.visibility,
        isLeaf <- source.isLeaf,
        isAbstract <- source.isAbstract,
        eAnnotations <- source.eAnnotations,
        ownedComment <- source.ownedComment,
        clientDependency <- source.clientDependency,
        nameExpression <- source.nameExpression,
        elementImport <- source.elementImport,
        packageImport <- source.packageImport,
        ownedRule <- source.ownedRule,
        templateParameter <- source.templateParameter,
        templateBinding <- source.templateBinding,
        ownedTemplateSignature <- source.ownedTemplateSignature,
        generalization <- source.generalization,
        powertypeExtent <- source.powertypeExtent,
        redefinedClassifier <- source.redefinedClassifier,
        substitution <- source.substitution,
        representation <- source.representation,
        collaborationUse <- source.collaborationUse,
        ownedUseCase <- source.ownedUseCase,
        useCase <- source.useCase,
        ownedAttribute <- source.ownedAttribute,
        ownedOperation <- source.ownedOperation,
        ownedLiteral <- source.ownedLiteral)
        do{
            for(as in source.getApplicableStereotypes()){
                if(as.name = 'enum' and target.hasStereotype('enum') = false){
                        target.applyStereotype(as);
                }
            }
        }
}
```

Transformation from PIM Enum Literal to Ethereum PSM Enum Member

```
rule EnumerationLiteral {
    from s : UML!"uml::EnumerationLiteral" in IN
    using {
        target : UML!"uml::EnumerationLiteral" = source.resolve();
    }

    to t : UML!"uml::EnumerationLiteral" (
        name <- source.name,
        visibility <- source.visibility,
        eAnnotations <- source.eAnnotations,
        ownedComment <- source.ownedComment,
        clientDependency <- source.clientDependency,
        nameExpression <- source.nameExpression,
        deployment <- source.deployment,
        templateParameter <- source.templateParameter,
        classifier <- source.classifier,
        slot <- source.slot,
        specification <- source.specification)
        do{
            for(as in source.getApplicableStereotypes()){
                if(as.name = 'member' and target.hasStereotype('member') = false){
                        target.applyStereotype(as);
                }
            }
        }
}
```

Transformation from PIM Property to Ethereum PSM Variable Property

```
rule Property2Variable {
    from
        source: UML!"uml::Property" in IN
    using {
        target : UML!"uml::Property" = source.resolve();
    }

    to
        t: UML!"uml::Property" (
            aggregation <- source.aggregation,
            association <- source.association,
            defaultValue <- source.defaultValue,
            deployment <- source.deployment,
            eAnnotations <- source.eAnnotations,
            isDerived <- source.isDerived,
            isDerivedUnion <- source.isDerivedUnion,
            isLeaf <- source.isLeaf,
            isOrdered <- source.isOrdered,
            isReadOnly <- source.isReadOnly,
            isStatic <- source.isStatic,
            isUnique <- source.isUnique,
            lowerValue <- source.lowerValue,
            name <- source.name,
            nameExpression <- source.nameExpression,
            ownedComment <- source.ownedComment,
            qualifier <- source.qualifier,
            redefinedProperty <- source.redefinedProperty,
            subsettedProperty <- source.subsettedProperty,
            templateParameter <- source.templateParameter,
            type <- source.type,
            upperValue <- source.upperValue,
            visibility <- source.visibility
        )
        do{
            if(source.hasStereotype('SmartContract')) {
                for(as in source.getApplicableStereotypes()){
                    if(as.name = 'variable' and target.hasStereotype('variable') = false){
                        target.applyStereotype(as);
                    }
                }
            }
        }
}
```

Transformation from PIM Operations to Ethereum PSM Functions

```
rule Operation {
    from
        source: UML!"uml::Operation" in IN
    using {
        target : UML!"uml::Operation" = source.resolve();
    }
    to
        t: UML!"uml::Operation" (
            bodyCondition <- source.bodyCondition,
            concurrency <- source.concurrency,
            eAnnotations <- source.eAnnotations,
            elementImport <- source.elementImport,
            isAbstract <- source.isAbstract,
            isLeaf <- source.isLeaf,
            isQuery <- source.isQuery,
            isStatic <- source.isStatic,
            method <- source.method,
            name <- source.name,
            nameExpression <- source.nameExpression,
            ownedComment <- source.ownedComment,
            ownedParameter <- source.ownedParameter,
            ownedParameterSet <- source.ownedParameterSet,
            ownedRule <- source.ownedRule,
            ownedTemplateSignature <- source.ownedTemplateSignature,
            packageImport <- source.packageImport,
            postcondition <- source.postcondition,
            precondition <- source.precondition,
            raisedException <- source.raisedException,
            redefinedOperation <- source.redefinedOperation,
            templateBinding <- source.templateBinding,
            templateParameter <- source.templateParameter,
            visibility <- source.visibility
            )
            do{
                if(source.isStereotypeApplied().oclIsUndefined()) {
                    for(as in source.getApplicableStereotypes()){
                        if(as.name = 'function' and target.hasStereotype('function') = false){
                            target.applyStereotype(as);
                        }
                    }
                }
            }
}
```

Transformation from PIM StateMachine to Ethereum PSM StateMachine

```
rule StateMachine {
    from source : UML!"uml::StateMachine", r: UML!Region in IN
(source.oclIsTypeOf(UML!"uml::StateMachine"))
    to t : UML!"uml::StateMachine" (
        name <- source.name,
        visibility <- source.visibility,
        isLeaf <- source.isLeaf,
        isAbstract <- source.isAbstract,
        isActive <- source.isActive,
        isReentrant <- source.isReentrant,
        eAnnotations <- source.eAnnotations,
        ownedComment <- source.ownedComment,
        clientDependency <- source.clientDependency,
        nameExpression <- source.nameExpression,
        elementImport <- source.elementImport,
        packageImport <- source.packageImport,
        ownedRule <- source.ownedRule,
        templateParameter <- source.templateParameter,
        templateBinding <- source.templateBinding,
        ownedTemplateSignature <- source.ownedTemplateSignature,
        generalization <- source.generalization,
        powertypeExtent <- source.powertypeExtent,
        redefinedClassifier <- source.redefinedClassifier,
        substitution <- source.substitution,
        representation <- source.representation,
        collaborationUse <- source.collaborationUse,
        ownedUseCase <- source.ownedUseCase,
        useCase <- source.useCase,
        ownedAttribute <- source.ownedAttribute,
        ownedConnector <- source.ownedConnector,
        ownedBehavior <- source.ownedBehavior,
        classifierBehavior <- source.classifierBehavior,
        interfaceRealization <- source.interfaceRealization,
        ownedTrigger <- source.ownedTrigger,
        nestedClassifier <- source.nestedClassifier,
        ownedOperation <- source.ownedOperation,
        ownedReception <- source.ownedReception,
        redefinedBehavior <- source.redefinedBehavior,
        ownedParameter <- source.ownedParameter,
        precondition <- source.precondition,
        postcondition <- source.postcondition,
        ownedParameterSet <- source.ownedParameterSet,
        specification <- source.specification,
        region <- source.region,
        submachineState <- source.submachineState,
        connectionPoint <- source.connectionPoint,
        extendedStateMachine <- source.extendedStateMachine),
       stateMachine2Enum: UML!Enumeration (
           name <- source.name+'State',
           ownedLiteral <- Set{stateliteral}
       ),
       state2literal: distinct UML!EnumerationLiteral foreach(st in r.subvertex ->
               select(e | e.oclIsTypeOf(UML!State))) (
           name <- st.name
       )
}
```

Transformation from PIM Effects to Ethereum PSM Event Operations

```
rule Effect2Event {
    from source : UML!"uml::Transition" in IN (source.oclIsTypeOf(UML!"uml::Transition"))
        using {
            event : UML!"uml::Operation" = te.resolve()
        }
    to t : UML!"uml::Transition" (
        __xmiID__ <- source.__xmiID__,
        name <- source.name,
        visibility <- source.visibility,
        isLeaf <- source.isLeaf,
        kind <- source.kind,
        eAnnotations <- source.eAnnotations,
        ownedComment <- source.ownedComment,
        clientDependency <- source.clientDependency,
        nameExpression <- source.nameExpression,
        elementImport <- source.elementImport,
        packageImport <- source.packageImport,
        ownedRule <- source.ownedRule,
        source <- source.source,
        target <- source.target,
        redefinedTransition <- source.redefinedTransition,
        guard <- source.guard,
        effect <- source.effect,
        trigger <- source.trigger),
       event: distinct UML!Operation foreach(te in r.transition) (
            name <- te.effect.name,
            ownedParameter <- te.effect.ownedParameter,
            ownedParameterSet <- te.effect.ownedParameterSet,
            visibility <- #public
       )
       do{
            for(as in source.getApplicableStereotypes()){
                if(as.name = 'event' and target.hasStereotype('event') = false){
                    target.applyStereotype(as);
                    }
                }
            }
       }
}
```

Appendix 3. Blockchain PIM to Hyperledger Fabric PSM transformation rules

Transformation from PIM SmartContract to Hyperledger Fabric PSM Chaincode

```
rule SmartContract2Chaincode {
    from
    source: UML!"uml::Class" in IN (
        source.oclIsTypeOf(UML!"uml::Class")
    )
    using {
        target : UML!"uml::Class" = source.resolve();
    }
    to
    t: UML!"uml::Class" (
        name <- source.name,
        visibility <- source.visibility,
        isLeaf <- source.isLeaf,
        isAbstract <- source.isAbstract,
        isActive <- source.isActive,
        eAnnotations <- source.eAnnotations,
        ownedComment <- source.ownedComment,
        nameExpression <- source.nameExpression,
        elementImport <- source.elementImport,
        packageImport <- source.packageImport,
        ownedRule <- source.ownedRule,
        templateParameter <- source.templateParameter,
        templateBinding <- source.templateBinding,
        ownedTemplateSignature <- source.ownedTemplateSignature,
        generalization <- source.generalization,
        powertypeExtent <- source.powertypeExtent,
        redefinedClassifier <- source.redefinedClassifier,
        substitution <- source.substitution,
        representation <- source.representation,
        collaborationUse <- source.collaborationUse,
        ownedUseCase <- source.ownedUseCase,
        useCase <- source.useCase,
        ownedAttribute <- source.ownedAttribute,
        ownedConnector <- source.ownedConnector,
        ownedBehavior <- source.ownedBehavior,
            classifierBehavior <- source.classifierBehavior,
        interfaceRealization <- source.interfaceRealization,
        nestedClassifier <- source.nestedClassifier,
        ownedOperation <- source.ownedOperation,
        ownedReception <- source.ownedReception
    )
    do{
        if(source.hasStereotype('SmartContract')) {
            for(as in source.getApplicableStereotypes()){
                if(as.name = 'chaincode' and target.hasStereotype('chaincode') = false){
                    target.applyStereotype(as);
                    for (a in as.getAllAttributes()) {
                        if (not a.name.startsWith('base_Class') and source.hasValue(as, a.name))
{
                            target.setValue(as, a.name, source.getValue(as, a.name));
                        }
                    }
                }
            }
        }
    }
}
```

Transformation from PIM Contained Classes to Hyperledger Fabric PSM Structure Class

```
rule ContainedClass2Structure {
   from
    source: UML!"uml::Class" in IN (
        source.oclIsTypeOf(UML!"uml::Class")
    )
    using {
        target : UML!"uml::Class" = source.resolve();
    }
   to
    t: UML!"uml::Class" (
        name <- source.name,
        visibility <- source.visibility,
        isLeaf <- source.isLeaf,
        isAbstract <- source.isAbstract,
        isActive <- source.isActive,
        eAnnotations <- source.eAnnotations,
        ownedComment <- source.ownedComment,
        nameExpression <- source.nameExpression,
        elementImport <- source.elementImport,
        packageImport <- source.packageImport,
        ownedRule <- source.ownedRule,
        templateParameter <- source.templateParameter,
        templateBinding <- source.templateBinding,
        ownedTemplateSignature <- source.ownedTemplateSignature,
        generalization <- source.generalization,
        powertypeExtent <- source.powertypeExtent,
        redefinedClassifier <- source.redefinedClassifier,
        substitution <- source.substitution,
        representation <- source.representation,
        collaborationUse <- source.collaborationUse,
        ownedUseCase <- source.ownedUseCase,
        useCase <- source.useCase,
        ownedAttribute <- source.ownedAttribute,
        ownedConnector <- source.ownedConnector,
        ownedBehavior <- source.ownedBehavior,
            classifierBehavior <- source.classifierBehavior,
        interfaceRealization <- source.interfaceRealization,
        nestedClassifier <- source.nestedClassifier,
        ownedOperation <- source.ownedOperation,
        ownedReception <- source.ownedReception
    )
    do{
        if(source.owner.hasStereotype('SmartContract')) {
            for(as in source.getApplicableStereotypes()){
                if(as.name = 'structure' and target.hasStereotype('structure') = false){
                    target.applyStereotype(as);
                    for (a in as.getAllAttributes()) {
                        if (not a.name.startsWith('base_Class') and source.hasValue(as, a.name))
{
                            target.setValue(as, a.name, source.getValue(as, a.name));
                        }
                    }
                }
            }
        }
    }
}
```

Transformation from PIM Contained Class Properties to Hyperledger Fabric PSM Structure Class Field Properties

```
rule Property2Field {
    from
        source: UML!"uml::Property" in IN
        using {
            target : UML!"uml::Property" = source.resolve();
        }
    to
        t: UML!"uml::Property" (
            aggregation <- source.aggregation,
            association <- source.association,
            defaultValue <- source.defaultValue,
            deployment <- source.deployment,
            eAnnotations <- source.eAnnotations,
            isDerived <- source.isDerived,
            isDerivedUnion <- source.isDerivedUnion,
            isLeaf <- source.isLeaf,
            isOrdered <- source.isOrdered,
            isReadOnly <- source.isReadOnly,
            isStatic <- source.isStatic,
            isUnique <- source.isUnique,
            lowerValue <- source.lowerValue,
            name <- source.name,
            nameExpression <- source.nameExpression,
            ownedComment <- source.ownedComment,
            qualifier <- source.qualifier,
            redefinedProperty <- source.redefinedProperty,
            subsettedProperty <- source.subsettedProperty,
            templateParameter <- source.templateParameter,
            type <- source.type,
            upperValue <- source.upperValue,
            visibility <- source.visibility
        )
        do{
            if(source.isStereotypeApplied().oclIsUndefined()) {
                for(as in source.getApplicableStereotypes()){
                    if(as.name = 'field' and target.hasStereotype('field') = false){
                        target.applyStereotype(as);
                    }
                }
            }
        }
}
```

Transformation from PIM Property to Hyperledger Fabric PSM Variable Property

```
rule Property2Variable {
    from
        source: UML!"uml::Property" in IN
        using {
            target : UML!"uml::Property" = source.resolve();
        }
    to
        t: UML!"uml::Property" (
            aggregation <- source.aggregation,
            association <- source.association,
            defaultValue <- source.defaultValue,
            deployment <- source.deployment,
            eAnnotations <- source.eAnnotations,
            isDerived <- source.isDerived,
            isDerivedUnion <- source.isDerivedUnion,
            isLeaf <- source.isLeaf,
            isOrdered <- source.isOrdered,
            isReadOnly <- source.isReadOnly,
            isStatic <- source.isStatic,
            isUnique <- source.isUnique,
            lowerValue <- source.lowerValue,
            name <- source.name,
            nameExpression <- source.nameExpression,
            ownedComment <- source.ownedComment,
            qualifier <- source.qualifier,
            redefinedProperty <- source.redefinedProperty,
            subsettedProperty <- source.subsettedProperty,
            templateParameter <- source.templateParameter,
            type <- source.type,
            upperValue <- source.upperValue,
            visibility <- source.visibility
        )
        do{
            if(source.hasStereotype('SmartContract')) {
                for(as in source.getApplicableStereotypes()){
                    if(as.name = 'variable' and target.hasStereotype('variable') = false){
                        target.applyStereotype(as);
                    }
                }
            }
        }
}
```

Transformation from PIM Operations to Hyperledger Fabric PSM Functions

```
rule Operation {
    from
        source: UML!"uml::Operation" in IN
            using {
                    target : UML!"uml::Operation" = source.resolve();
            }
    to
        t: UML!"uml::Operation" (
            bodyCondition <- source.bodyCondition,
            concurrency <- source.concurrency,
            eAnnotations <- source.eAnnotations,
            elementImport <- source.elementImport,
            isAbstract <- source.isAbstract,
            isLeaf <- source.isLeaf,
            isQuery <- source.isQuery,
            isStatic <- source.isStatic,
            method <- source.method,
            name <- if(source.visibility = #public) then source.name.toU1Case else source.name
endif,
            nameExpression <- source.nameExpression,
            ownedComment <- source.ownedComment,
            ownedParameter <- source.ownedParameter,
            ownedParameterSet <- source.ownedParameterSet,
            ownedRule <- source.ownedRule,
            ownedTemplateSignature <- source.ownedTemplateSignature,
            packageImport <- source.packageImport,
            postcondition <- source.postcondition,
            precondition <- source.precondition,
            raisedException <- source.raisedException,
            redefinedOperation <- source.redefinedOperation,
            templateBinding <- source.templateBinding,
            templateParameter <- source.templateParameter,
            visibility <- source.visibility
            )
            do{
                if(source.isStereotypeApplied().oclIsUndefined()) {
                    for(as in source.getApplicableStereotypes()){
                        if(as.name = 'function' and target.hasStereotype('function') = false){
                            target.applyStereotype(as);
                        }
                    }
                }
            }
}
```

Transformation from PIM Effects to Hyperledger Fabric PSM Structure Class

```
rule Effect2Struct {
    from source : UML!"uml::Transition" in IN (source.oclIsTypeOf(UML!"uml::Transition"))
    using {
        struct : UML!"uml::Class" = te.resolve()
    }
    to t : UML!"uml::Transition" (
        __xmiID__ <- source.__xmiID__,
        name <- source.name,
        visibility <- source.visibility,
        isLeaf <- source.isLeaf,
        kind <- source.kind,
        eAnnotations <- source.eAnnotations,
        ownedComment <- source.ownedComment,
        clientDependency <- source.clientDependency,
        nameExpression <- source.nameExpression,
        elementImport <- source.elementImport,
        packageImport <- source.packageImport,
        ownedRule <- source.ownedRule,
        source <- source.source,
        target <- source.target,
        redefinedTransition <- source.redefinedTransition,
        guard <- source.guard,
        effect <- source.effect,
        trigger <- source.trigger),
    struct: distinct UML!Class foreach(te in r.transition) (
        name <- te.effect.name,
        ownedAttribute <- te.effect.ownedParameter)
    do{
        for(as in source.getApplicableStereotypes()){
            if(as.name = 'struct' and target.hasStereotype('struct') = false){
                target.applyStereotype(as);
                }
            }
        }
    }
}
```

Appendix 4. Ethereum PSM to Solidity smart contract transformation templates

Ethereum PSM to Solidity contract

```
[template public EthereumPSMtoSolidity(model : Model)]
[file (model.name.concat('.sol'), false)]
// SPDX-License-Identifier: MIT
pragma solidity >=0.4.0 <0.8.0;
contract [model.ownedElement->filter(Class).name/] [if
(model.ownedElement-
>filter(Class).isStereotypeApplied(contract).superClass->notEmpty())] is
[model.ownedElement->filter(Class).superClass.name/] [/if]{
[for (e : Element | model.ownedElement)]
[let c : Class = e]
             [for (p : Property | c.ownedAttribute)]
             [p.type.name/] [p.visibility/] [p.name/] [if
(p.isSetDefault())] = [p.default/] [/if];
             [/for]
             [for (cn : Class | c.ownedElement->filter(Class))]
             struct [cn.name/]{
                          [for (p : Property | cn.attribute) ]
                          [p.type.name/] [p.name/];
                          [/for]
             }
             [/for]
             [for (en : Enumeration | c.ownedElement-
>filter(Enumeration))]
             enum [en.name/]{
                          [for (el : EnumerationLiteral |
en.ownedLiteral) separator(', ')]
                                       [el.name/]
                          [/for]
             }
             [/for]
             [for (m : Constraint | c.ownedElement->filter(Constraint))]
             modifier [m.name/](){
                          [m.specification/]
             }
             [/for]
             [for (sm : StateMachine | c.ownedBehavior-
>filter(StateMachine))][for (r : Region | sm.region)][for (it :
Transition | r.transition)][for (x : Trigger | it.trigger)][let var :
CallEvent = x.event]
             [if (r.transition.trigger.event->exists(e: Event |
e.oclAsType(CallEvent).operation = var.operation)._not())]
             [for (o : Operation | c.ownedOperation)]
             [if (o.isStereotypeApplied(Event))]event[elseif
(o.isStereotypeApplied(Constructor))]constructor[elseif
(o.isStereotypeApplied(Function))]function[/if] [o.name/] ([for (pr :
Parameter | o.ownedParameter) separator(', ') ?(not(pr.direction =
ParameterDirectionKind::return))][pr.type.name/] [pr.name/][/for])
[o.visibility/] [for (mod : Constraint | o.ownedRule)] [mod.name/]
[/for][if (o.ownedParameter->any(p | p.direction =
ParameterDirectionKind::return)->notEmpty())]returns ([o.ownedParameter-
>last().type.name/])[/if] {
             [for (b : Behavior | o.method)]
                          [let it : Interaction = b]
[interactionFragment2Expression(it.fragment)/][/let]
                          [let ob : OpaqueBehavior = b] [ob._body/][/let]
             [/for]
             }
             [/for][/if][/let][/for][/for][/for][/for]
             [stateMachinetoFunctions(c)/]
[/let]
[/for]
}[/file]
[/template]
```

Interaction to Function body

```
[template public interactionFragment2Expression(fragments:
OrderedSet(InteractionFragment))]
[for (inf : InteractionFragment | fragments)]
[let mos : MessageOccurrenceSpecification = self]
             [if (mos.message.receiveEvent.message =
mos.message.sendEvent.message)][mos(mos)/][/if]
[elselet  cf : CombinedFragment = self]
             [combinedFragment2Operands(cf)/]}
[/let][/for][/template]

[template public mos(mos: MessageOccurrenceSpecification)]
[if (mos.message.messageSort = MessageSort::reply and
mos.message.receiveEvent.oclIsKindOf(Gate))]return [mos.message.name/];[elseif
(mos.message.messageSort = MessageSort::createMessage and
mos.message.receiveEvent = mos)][let m: MessageOccurrenceSpecification =
mos.message.sendEvent][mos.covered.represents.type.name/]
[mos.covered.represents.name/] [let p: Parameter = m.covered.represents][if
(p.isMultivalued())][ '[' /][ ']' /][/if][/let];[/let][elseif
(mos.message.messageSort = MessageSort::synchCall and mos.message.receiveEvent =
mos and
not(mos.message.sendEvent.oclIsKindOf(Gate)))][sm(mos.message)/][/if][/template]
[template public sm(m : Message)]
[let op : Operation = m.signature]
[if (m.argument->size() = op.ownedParameter->size())][if (op.ownedParameter-
>any(p | p.direction = ParameterDirectionKind::return)->notEmpty())][m.argument-
>last().stringValue()/] = [/if][op.name/]([for (oe : ValueSpecification |
m.argument) separator(', ') ? (not(op.ownedParameter->at(m.argument-
>indexOf(oe)).direction = ParameterDirectionKind::return))
][oe.stringValue()/][/for]);
[else][if (op.ownedParameter->any(p | p.direction =
ParameterDirectionKind::return)->notEmpty())][op.ownedParameter->last().name/] =
[/if][op.name/]([for (pr : Parameter | op.ownedParameter) separator(', ') ?
(not(self.direction = ParameterDirectionKind::return)) ][pr.name/][/for]);[/if]
[else][m.name/];[/let]
[/template]
[template public alt2if(operand : InteractionOperand)]
[if (operand.precedingSiblings()-
>isEmpty())]if([operand.guard.specification.stringValue()/]){[elseif
(operand.followingSiblings()->isEmpty())]} else {[else]} else
if([operand.guard.specification.stringValue()/]){[/if][/template]
[template public loop2for(operand : InteractionOperand, int : Integer)]
[if (operand.guard.specification->notEmpty())]while
([operand.guard.specification.stringValue()/])
[elseif (operand.guard.minint->notEmpty())]for (var ['i'+int/] = 0; ['i'+int/]
<([operand.guard.minint.stringValue()/]); ['i'+int/]++)
[/if]{[/template]
[template public opt2if(operand : InteractionOperand)]
if([operand.guard.specification.stringValue()/]){[/template]
```

StateMachine to Functions

```
[template public stateMachinetoFunctions(c : Class)]
[let sm : StateMachine = c.classifierBehavior][for (r : Region | sm.region)][for (it : Transition |
r.transition)][for (x : Trigger | it.trigger)][let var : CallEvent = x.event]
function [var.operation.name/] ([for (pr : Parameter | var.operation.ownedParameter) separator(', ')
?(not(pr.direction = ParameterDirectionKind::return))][pr.type.name/] [pr.name/][/for])
[var.operation.visibility/] [for (mod : Constraint | var.operation.ownedRule)] [mod.name/] [/for][if
(var.operation.ownedParameter->any(p | p.direction = ParameterDirectionKind::return)-
>notEmpty())]returns ([var.operation.ownedParameter->last().type.name/])[/if] {
{
[r.transition->reject(r.transition.trigger.event->exists(e: Event | e.oclAsType(CallEvent).operation =
var.operation))->size()/]
[ExtendOperation(it.container.transition->select(it.container.transition.trigger.event->exists(e: Event
| e.oclAsType(CallEvent).operation = var.operation)))/]
[/let][/for][/for][/for][/let]
[/template]
[query public sin(arg : String) : Integer = arg.indexOf('return')/]
[template public ExtendOperation(set : Set(Transition))]
[for (it : Transition | set)]
[for (s : Trigger | it.trigger)][let c : CallEvent = s.event]
[for (var : Vertex | set.source)]
[let s : State = var]
require ([c.operation.owner.name/].state==[s.name/] [if (it.guard->notEmpty())] and
[it.guard.specification/][/if] );
[elselet p : Pseudostate = var]
             [if (p.kind = PseudostateKind::junction)]
require ([for (ct : Transition | p.incoming) separator('or ')]
             ([c.operation.owner.name/].state=[ct.source.name/] [if (ct.guard->notEmpty())] and
[ct.guard.specification/][/if])
             [/for]);
             [elseif (p.kind = PseudostateKind::initial)][/if]
[/let]
[/for]
[for (var : Behavior | c.operation.method)]
[let ob : OpaqueBehavior = var]
[ob._body->first().substring(1, sin(ob._body->first())-1)/]
[/let]
[/for]
[for (var : Vertex | set.target)]
[if (set.target->count(Vertex) > 1)]
if ([c.operation.namespace/].state==[it.source.name/] [if (it.guard->notEmpty())] and
[it.guard.specification/][/if])
[/if]
[let p : Pseudostate = var]
             [if (p.kind = PseudostateKind::junction or p.kind = PseudostateKind::choice)]
                          [for (ct : Transition | p.outgoing)]
                          [if (ct.precedingSiblings()-
>isEmpty())]if([ct.guard.specification.stringValue()/]){[elseif (ct.followingSiblings()->isEmpty())]}
else {[else]} else if([ct.guard.specification.stringValue()/]){[/if]
                                       [if (ct.effect->notEmpty() and
ct.effect.oclIsKindOf(OpaqueBehavior))] emit [ct.effect.specification/][/if]
                                       [if (not(ct.target=it.source))]
[c.operation.owner.name/].state=[ct.target.name/][/if]
                          [/for]
             [elseif (p.kind = PseudostateKind::initial)]
             [/if]
[elselet s : State = var]
             [if (it.effect->notEmpty() and it.effect.oclIsKindOf(OpaqueBehavior))] emit
[it.effect.specification/][/if]
             [if (not(it.target=it.source))] [c.operation.owner.name/].state=[it.target.name/][/if]
[/let]
[/for]
[for (var : Behavior | c.operation.method)]
[let ob : OpaqueBehavior = var]
[ob._body->first().substring(sin(ob._body->first()))/]
[/let]
[/for]
[/let]
[/for]
[/for]
[/template]
```

Appendix 5. Hyperledger Fabric PSM to Go chaincode transformation templates

HyperledgerFabricPSM to Chaincode

```
[template public HyperledgerFabricPSMtoGo(model : Model)]
[file (model.name.concat('.go'), false)]
package chaincode
import (
		"encoding/json"
		"github.com/hyperledger/fabric-contract-api-go/contractapi"
)
[for (e : Element | model.ownedElement)][let c : Class = e]
type [c.name/] struct {
		contractapi.Contract
}
[for (p : Property | c.ownedAttribute)]
[if (p.isStereotypeApplied(Constant))]
const [p.type.name/] [p.visibility/] = [p.default/]
[elseif (p.isStereotypeApplied(Variable))]
var [p.name/] [p.type.name/] [if (p.isSetDefault())] = [p.default/] [/if]
[/if]
[/for]
[for (cn : Class | c.ownedElement->filter(Class))]
type [cn.name/] struct{
		[for (p : Property | cn.attribute)]
		[p.name/] [p.type.name/] 'json:"[p.name/]"'
		[/for]
}
[/for]
[for (sm : StateMachine | c.ownedBehavior->filter(StateMachine))][for (r
: Region | sm.region)][for (it : Transition | r.transition)][for (x :
Trigger | it.trigger)][let var : CallEvent = x.event]
[if (r.transition.trigger.event->exists(e: Event |
e.oclAsType(CallEvent).operation = var.operation)._not())]
[for (o : Operation | c.ownedOperation)]
func (s *[c.name/]) [o.name/] ([for (pr : Parameter | o.ownedParameter)
separator(', ') ?(not(pr.direction =
ParameterDirectionKind::return))][pr.name/] [pr.type.name/][/for]) [for
(pr : Parameter | o.ownedParameter)  ?((pr.direction =
ParameterDirectionKind::return))][pr.type.name/][/for] {
		[for (b : Behavior | o.method)]
		[let it : Interaction = b]
[interactionFragment2Expression(it.fragment)/]
		[elselet  it : OpaqueBehavior = b][it._body/][/let]
		[/for]
[/for]
}
[/if]
[/let][/for][/for][/for][/for]
[stateMachinetoFunctions(c)/]
[if (c.isStereotypeApplied(Chaincode).superClass-
>notEmpty())][c.superClass.toString()/] [/if][/let]
[/for]
[/file]
[/template]
```

Interaction to Function Body

```
[template public combinedFragment2Operands(cf : CombinedFragment)]
[for (io : InteractionOperand | cf.operand)]
[if (cf.interactionOperator = InteractionOperatorKind::alt)][altf(io)/]
[elseif (cf.interactionOperator = InteractionOperatorKind::loop)][loopf(io,
cf.ancestors(CombinedFragment)->select(a: CombinedFragment | a.interactionOperator =
InteractionOperatorKind::loop)->size())/]
[elseif (cf.interactionOperator = InteractionOperatorKind::opt)][optf(io)/]
[/if][interactionFragment2Expression(io.fragment)/][/for][/template]

[template public interactionFragment2Expression(fragments:
OrderedSet(InteractionFragment))]
[for (inf : InteractionFragment | fragments)]
[let mos : MessageOccurrenceSpecification = self]
             [if (mos.message.receiveEvent.message =
mos.message.sendEvent.message)][mos(mos)/][/if]
[elselet  cf : CombinedFragment = self]
             [combinedFragment2Operands(cf)/]}
[/let][/for][/template]

[template public mos(mos: MessageOccurrenceSpecification)]
[if (mos.message.messageSort = MessageSort::reply and
mos.message.receiveEvent.oclIsKindOf(Gate))]stub.return("[mos.message.name/]")[elseif
(mos.message.messageSort = MessageSort::createMessage)][let m:
MessageOccurrenceSpecification = mos.message.receiveEvent]var
[mos.covered.represents.name/] [let p: Parameter = m.covered.represents] [if
(p.isMultivalued())][ '[' /][ ']' /][/if][/let]
[mos.covered.represents.type.name/][/let][elseif (mos.message.messageSort =
MessageSort::synchCall and mos.message.receiveEvent = mos and
not(mos.message.sendEvent.oclIsKindOf(Gate)))][sm(mos.message)/][/if][/template]
[template public sm(m : Message)]
[let op : Operation = m.signature]
[if (m.argument->size() = op.ownedParameter->size())][if (op.ownedParameter->any(p |
p.direction = ParameterDirectionKind::return)->notEmpty())][m.argument-
>last().stringValue()/] := [/if][op.name/]([for (oe : ValueSpecification |
m.argument) separator(', ') ? (not(op.ownedParameter->at(m.argument-
>indexOf(oe)).direction = ParameterDirectionKind::return))
][oe.stringValue()/][/for])
if err != nil {
                          return nil, err
}[else][if (op.ownedParameter->any(p | p.direction = ParameterDirectionKind::return)-
>notEmpty())][op.ownedParameter->last().name/] := [/if][op.name/]([for (pr :
Parameter | op.ownedParameter) separator(', ') ? (not(self.direction =
ParameterDirectionKind::return)) ][pr.name/][/for])
if err != nil {
             return err
}[/if]
[else][m.name/][/let]
[/template]
[template public altf(operand : InteractionOperand)]
[if (operand.precedingSiblings()-
>isEmpty())]if([operand.guard.specification.stringValue()/]){[elseif
(operand.followingSiblings()->isEmpty())]}else{[else]}else
if([operand.guard.specification.stringValue()/]){[/if][/template]
[template public loopf(operand : InteractionOperand, int : Integer)]
[if (operand.guard.specification->notEmpty())]for
([operand.guard.specification.stringValue()/])[elseif (operand.guard.minint-
>notEmpty())]for ['i'+int/] := 0; ['i'+int/] <[operand.guard.minint.stringValue()/]
['i'+int/]++[/if]{[/template]
[template public optf(operand : InteractionOperand)]
if([operand.guard.specification.stringValue()/]){[/template]
[template public breakf(operand : InteractionOperand)]
if([operand.guard.specification.stringValue()/]){
[interactionFragment2Expression(operand.fragment)/][/template]
```

StateMachine to Functions

```
[template public stateMachinetoFunctions(c : Class)]
[let sm : StateMachine = c.classifierBehavior][for (r : Region | sm.region)][for (it : Transition |
r.transition)][for (x : Trigger | it.trigger)][let var : CallEvent = x.event]
func     (s     *[var.operation.name/])     [var.operation.name/]     ([for     (pr     :     Parameter     |
var.operation.ownedParameter)           separator(',           ')           ?(not(pr.direction           =
ParameterDirectionKind::return))][pr.name/]    [pr.type.name/][/for])    [for    (pr    :    Parameter    |
var.operation.ownedParameter)  ?((pr.direction = ParameterDirectionKind::return))][pr.type.name/][/for]
{
{
[r.transition->reject(r.transition.trigger.event->exists(e:  Event  |  e.oclAsType(CallEvent).operation  =
var.operation))->size()/]
[ExtendOperation(it.container.transition->select(it.container.transition.trigger.event->exists(e:  Event
| e.oclAsType(CallEvent).operation = var.operation)))/]
[/let][/for][/for][/for][/let]
[/template]
[query public sin(arg : String) : Integer = arg.indexOf('return')/]
[template public ExtendOperation(set : Set(Transition))]
[for (it : Transition | set)]
[for (s : Trigger | it.trigger)][let c : CallEvent = s.event]
[for (var : Vertex | set.source)]
[let s : State = var]
if        [c.operation.namespace/].State=="[s.name/]"        [if        (it.guard->notEmpty())]        &&
[it.guard.specification/][/if]
[elselet p : Pseudostate = var]
             [if (p.kind = PseudostateKind::junction)]
if ([for (ct : Transition | p.incoming) separator('|| ')]
             [c.operation.namespace/].State=="[ct.source.name/]"    [if    (ct.guard->notEmpty())]    &&
[ct.guard.specification/][/if]
             [/for])
             [elseif (p.kind = PseudostateKind::initial)][/if]
[/let]
[/for]
[for (var : Behavior | c.operation.method)]
[let ob : OpaqueBehavior = var]
[ob._body->first().substring(1, sin(ob._body->first())-1)/]
[/let]
[/for]
[for (var : Vertex | set.target)]
[if (set.target->count(Vertex) > 1)]
if      [c.operation.namespace/].State=="[it.source.name/]"      [if      (it.guard->notEmpty())]      &&
[it.guard.specification/][/if]
[/if]
[let p : Pseudostate = var]
[if (p.kind = PseudostateKind::junction or p.kind = PseudostateKind::choice)]
             [for (ct : Transition | p.outgoing)]
             [if                                                           (ct.precedingSiblings()-
>isEmpty())]if([ct.guard.specification.stringValue()/]){[elseif    (ct.followingSiblings()->isEmpty())]}
else {[else]} else if([ct.guard.specification.stringValue()/]){[/if]
             [if        (ct.effect->notEmpty()        and        ct.effect.oclIsKindOf(OpaqueBehavior))]
[ct.effect.specification/][/if]
             [if (not(ct.target=it.source))] [c.operation.namespace/].State="[ct.target.name/]"[/if]
             [/for]
             [elseif (p.kind = PseudostateKind::initial)]
             [/if]
[elselet s : State = var]
             [if        (it.effect->notEmpty()        and        it.effect.oclIsKindOf(OpaqueBehavior))]
[it.effect.specification/][/if]
             [if (not(it.target=it.source))] [c.operation.namespace/].State="[it.target.name/]"[/if]
[/let]
[/for]
[for (var : Behavior | c.operation.method)]
[let ob : OpaqueBehavior = var][ob._body->first().substring(sin(ob._body->first()))/][/let]
[/for]
[/let][/for][/for]
[/template]
```

UDK 004.415.2+004.45](043.3)

SL 344. 2023-05-15, 21,25 leidyb. apsk. l. Tiražas 14 egz. Užsakymas 170.
Išleido Kauno technologijos universitetas, K. Donelaičio g. 73, 44249 Kaunas
Spausdino leidyklos „Technologija“ spaustuvė, Studentų g. 54, 51424 Kaunas