IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received June 22, 2020, accepted June 24, 2020, date of publication June 29, 2020, date of current version July 7, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3005519

Applying Model-Driven Engineering to Distributed Ledger Deployment

TOMASZ GÓRSKI, (Member, IEEE), AND JAKUB BEDNARSKI
Department of Computer Science, Polish Naval Academy of the Heroes of Westerplatte, 81-127 Gdynia, Poland
Corresponding author: Tomasz Górski (t.gorski@amw.gdynia.pl)

ABSTRACT Distributed Ledger Technology (DLT) enables data storage in a decentralized manner among collaborating parties. The software architecture of such solutions encompasses models placed in the relevant architectural views. A lot of research is devoted to smart contracts and consensus algorithms, which are realized by distributed applications and can be positioned within the *Logical* view. However, we see the need to provide modeling support for the *Deployment* view of distributed ledger solutions. Especially since the chosen DLT framework has a significant impact on implementation and deployment. Besides, consistency between models and configuration deployment scripts should be ensured. So, we have applied Model-Driven Engineering (MDE) that allows on the transformation of models into more detailed models, source code, or tests. We have proposed Unified Modeling Language (UML) stereotypes and tagged values for distributed ledger deployment modeling and placed them in the *UML Profile for Distributed Ledger Deployment*. We have also designed the *UML2Deployment* model-to-code transformation for the R3 Corda DLT framework. A UML Deployment model is the source whereas a Gradle Groovy deployment script is the target of the transformation. We have provided the complete solution by incorporating the transformation into the Visual Paradigm modeling tool. Furthermore, we have designed a dedicated plug-in to validate generated deployment scripts. In the paper, we have shown how to design transformation for generating deployment scripts for the R3 Corda DLT framework with the ability to switch to another one.

INDEX TERMS Distributed ledger, model-driven engineering, architectural views model $1+5$, deployment view, unified modeling language extensibility mechanisms.

I. INTRODUCTION

In a distributed ledger there is no central data store. Details of transactions among peers are stored in multiple places. A distributed ledger, defined by Xu *et al.* [1], is an append-only store of transactions, which is distributed across many machines. An append-only means that new transactions can be added but existing ones can not be modified or deleted. A new transaction might reverse a previous one, but both of them remain part of the ledger to allow audits and ensure integrity. The ledger is distributed and stored by the nodes of a peer-to-peer (P2P) network where each block is created at a predefined interval in a decentralized fashion employing a consensus algorithm that guarantees data integrity. A consensus algorithm is a primary element of DLT and is used to synchronize a distributed ledger at multiple nodes. There are four the most popular algorithms of finding consensus in distributed systems: practical byzantine fault tolerance (PBFT), proof-of-stake (PoS), delegated proof-of-stake (DPoS), and proof-of-work (PoW). In order to create a block, in the PoW consensus algorithm, a block creator (miner) must solve a cryptographic riddle to produce a hash [2]. That computationally complex task consumes a lot of electricity and causes delays. To overcome these problems, a proof-of-stake consensus mechanism has been developed recently, which enables achieving the consensus by proving the stake ownership [3]. The core idea of the PoS consensus algorithm evolves around the concept that the nodes willing to participate in the block creation process must prove that they own a certain number of coins at first and must lock a specified number of them, called stake.

A distributed ledger has at least the following set of features:

- distributed consensus on the ledger's state — capability to achieve a distributed consensus on the state of the ledger without being reliant on any trusted third party,

The associate editor coordinating the review of this manuscript and approving it for publication was Porfirio Tramontana.

- immutability and irreversibility of the ledger's state — achieving a distributed consensus ensures that the state of the ledger becomes practically immutable and irreversible after a certain period of time,
- data persistence — data is stored in a distributed fashion ensuring its persistency as long as there are participating nodes in the P2P network,
- data non-repudiation — every transaction needs to be digitally signed using cryptography public key which ensures the authenticity of the source of data,
- distributed data control — storing or retrieving data in/from the ledger can be carried out in a distributed manner that exhibits no single point of failure.

We can indicate two dominant distributed ledger types:

- public ledger — allows anyone to modify the ledger's state by storing new blocks and updating data by means of transactions among participating entities. The information stored in the ledger is transparent and accessible to everyone, which raises privacy concerns. It is also known as the permissionless ledger.
- private ledger — only authorized and trusted entities can participate in transactions within the ledger, which ensures the privacy of the ledger's data. It is also known as the permissioned ledger.

A blockchain is a distributed ledger that is structured into a linked list of blocks. Each block contains an ordered set of transactions. A block is linked to its predecessor using a cryptographic hash to secure the whole chain. Data stored in a blockchain can be verified even in a decentralized environment, which leads to numerous blockchain applications.

We can think of software architecture as a structure. Software architecture comprises software elements and relations among them. As far as software architecture is concerned, those elements exist at the highest level breakdown of a software system. We can look at the software system from different angles. In other words, we have different software architectural views. Kruchten [5] presents kind of reference model of software system with different architectural views: *Logical*, *Process*, *Physical*, *Development* and *Scenarios*. Another term for the *Scenarios* view is the *Use cases* view. Unified Modeling Language is the most commonly used graphical notation to model software elements. UML is especially useful for modeling systems designed under the object-oriented approach [6].

Transformations are central to Model-Driven Engineering, where they are used to transform models between different languages or different levels of abstraction. Kleppe *et al.* [7] provide the following definition of the Model-to-Model (M2M) transformation: Transformation is the automatic generation of a target model from a source model, according to a transformation definition. A transformation definition is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language. A transformation rule is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language (p. 24). Those that generate textual representation from models are called Model-to-Text (M2T) transformations. Besides, we can further divide M2T transformations into three subtypes: model-to-code (M2C), model-to-documentation (M2D), and as far as the quality of the software is concerned, model-to-test (M2Q) [8].

To be transformable, models have to be expressed in a modeling language. A model needs to be consistent with the metamodel, which defines the syntax and semantics of a certain type of model and is usually depicted in the UML class diagram. Mens and van Gorp [9] distinguish endogenous and exogenous transformations. Endogenous transformations occur between models expressed in the same modeling language. The latter transform models expressed in different modeling languages. Transformations can also be categorized as horizontal and vertical. A horizontal transformation occurs when both the source and the target models represent the same abstraction level. When they are at different levels of abstraction, we have a case of vertical transformation. Moreover, model transformations can be unidirectional or bidirectional. In the case of unidirectional transformation, the target model can be generated from the source model, whereas for bidirectional transformation, the source and the target models remain consistent. A traceability relationship is useful to establish links between elements of the models being transformed and help in checking the consistency [10].

The deployment environment provides conditions for running distributed applications. There is a lack of modeling means for distributed ledger solutions at the deployment level. Another important issue is the capability of modeling various types of DLT deployment environments, e.g.: dev, test, prod. For such a complex distributed solution with many deployment environments, we see the need to use MDE to support the automation of creating DLT deployment scripts with ensuring consistency of models and configuration files.

Our contribution consists of three elements. First of all, we have proposed new stereotypes, and tagged values, that describe the needed semantic UML enrichment for distributed ledger deployment. We have placed those elements in the *UML Profile for Distributed Ledger Deployment*. Secondly, we have designed *UML2Deployment* M2C transformation to automate the deployment of various runtime environments for distributed ledger solution. Thirdly, we have provided a consistency check between the UML Deployment model and deployment scripts. The proposed *UML2Deployment* transformation generates deployment scripts for DLT network configuration. The transformation is vertical, which means that the source and the target are at different levels of abstraction. The source of our transformation is the UML Deployment model enriched with stereotypes and tagged values from the proposed UML profile. Whereas, the target of our transformation is runnable Gradle script for DLT nodes deployment configuration. The transformation is also exogenous because the source and the target are expressed in different notations: UML and Groovy Domain Specific Language (DSL). In its current form, it is

also a unidirectional transformation. We have incorporated the transformation, as a plug-in, into the Visual Paradigm modeling tool.

The paper is structured as follows. Section 2 shows our model-driven approach in light of the current research. Section 3 locates the main area of interest of the paper within the *Architectural views model 1+5*. Section 4 describes the R3 Corda DLT framework. Section 5 contains a detailed description of dedicated *UML Profile for Distributed Ledger Deployment*. Section 6 presents the design of the *UML2Deployment* transformation that generates deployment scripts for DLT network configuration. According to our design principle, the implementation of the transformation was loosely coupled with the selected DLT framework. Section 7 reveals the options of the transformation usage on the example of the Electricity Consumption and Supply Management (ECSM) system [4]. Section 8 introduces assumptions for validation of our transformation and design of Java application for consistency checking between models and generated scripts. Section 9 encompasses discussion and limitations. The last one concludes the paper and outlines the direction of further work.

II. RELATED WORK

The paper focuses on distributed ledger (blockchain) technology and the Model-Driven Engineering approach. So we have searched for papers concerning those two issues. We divided the literature review results in three paragraphs. The first paragraph describes papers that treat blockchain technology. In the second one, we present articles that show recent advances in MDE. The third paragraph is focused on the research results of applying MDE to blockchain solutions.

Many scientists have paid attention to blockchain technology because it can bring benefits to various industries. Al-Jaroodi and Mohamed [11] and Monrat *et al.* [12] explore the advantages and challenges of incorporating blockchain in different industrial applications. The technology is a natural choice when designing supply chain solutions. Gonczol *et al.* [13] present the current state of research and summarize the benefits and the challenges of the distributed organization and management of supply chains. Whereas, Leng *et al.* [14] propose a public blockchain of Chinese agricultural supply chain system based on double chain architecture. The healthcare sector can greatly benefit from the distributed ledger technology due to decentralization and privacy. Shahnaz *et al.* [15] present a framework that could be used for the implementation of blockchain technology for the Electronic Health Record system. Ismail *et al.* [16] propose blockchain architecture for healthcare data management. They divide network participants into clusters and use the PBFT consensus algorithm, which has low energy demand. Many blockchain networks have adopted the PoW consensus mechanisms, in which the consensus is reached through the intensive mining process [2]. On the other hand, Nguyen *et al.* [3] investigate the PoS mechanisms, from fundamental to advanced ones along with performance analysis. Furthermore, we believe that blockchain has enormous potential in the military sector. Górski *et al.* [17] present a solution that persists warship position coordinates in the blockchain that is hosted in a cloud environment. In the energy sector, next-generation grid demands technologies, which enable the integration of distributed energy resources and consumers that both seamlessly buy and sell electricity. Wang *et al.* [18] developed an optimization model and blockchain-based architecture on IBM Hyperledger Fabric to manage the operation of crowdsourced energy systems, with peer-to-peer energy trading transactions. Lu *et al.* [19] propose a secure and efficient renewable energy trading mechanism based on blockchain that improves system availability. The authors of the paper have previously proposed software architecture of ECSM [4]. The ECSM has been designed on the R3 Corda distributed ledger platform [20]. A comparative analysis that evaluates the feasibility of the most well-established, both private and public DLT platforms, has been presented by Chowdhury *et al.* [21].

Model-Driven Engineering is also a field of recent studies. Because transformations play a significant role in MDE, practical methods are needed to help detect errors in transformations and automate their verification. Cuadrado *et al.* [22] present a method for the static analysis of ATL model transformations whereas Burgueno *et al.* [23] discuss static fault localization in model transformations. Guaranteeing the quality of early models is essential for a successful application of MDE techniques and related tool-supported model refinements. Autili *et al.* [24] propose a tool that derives a set of customizable questionnaires expressed in natural language from each model to be validated by domain experts. Macedo *et al.* [25] focus on inconsistency handling in MDE, particularly in model repair techniques. Consistency between the UML class model and its Java implementation is the field of study of Chavez *et al.* [26]. The quality and maintainability of model transformations are considered by Fleck *et al.* [27]. They propose an automated approach to modularize model transformations. Software models are usually expressed in Unified Modeling Language and may use Object Constraint Language (OCL) to provide precise semantics. Lu *et al.* [28] show application of OCL constraints to medical rules, in cancer registries, to ensure the quality of cancer data. The correctness of UML class diagrams annotated with OCL constraints can be checked using bounded verification techniques. Clariso *et al.* [29] present approach which may increase the usability of UML/OCL bounded verification tools and improve the efficiency of the verification process. Looking at recent applications of the MDE technique we can give an example of Núñez *et al.* [30]. They describe the MDE approach for mobile business applications focusing on the data layer. Yousaf *et al.* [8] propose an approach to test case generation for user interfaces from Interaction Flow Modeling Language (IFML) models. Moradi *et al.* [31] propose a framework that allows on services modeling, in a graphical environment, and transforming them into executable context web services. Magalhaes *et al.* [32] describe the iterative and

incremental process that guides model-to-model transformation development from requirements specification to implementation. Jacome and De Lara [33] propose a mechanism that allows specifying customization and extension rules for meta-models. Whereas, Hebig *et al.* [34] present a survey on approaches to support the co-evolution of metamodel and model.

Research results are also available for applying software engineering and model-driven engineering in designing blockchain solutions. Xu *et al.* [35] discuss two approaches for model-driven code generation. In the first, code of the smart contract is generated for collaborative business processes. The second application of MDE is a generator of blockchain registries for assets such as land titles, cars, or digital assets. They describe the *Regerator* tool which can generate and deploy smart contracts representing registries on the Ethereum blockchain. Górski and Bednarski [36] present the manner of smart contracts' modeling in blockchain solution. They propose the *UML Profile for Smart Contracts* and present static aspect of *Smart Contract Design Pattern*. Gao *et al.* [37] propose an automated approach and *SmartEmbed* prototype to clone or bug detection and validation of smart contracts.

Current research results focus mainly on smart contracts. We believe that the broader architectural description is advisable. Smart contracts are an essential part of distributed applications. Those applications must be installed on distributed ledger nodes, which make up the deployment environment. Thus, the paper concentrates on applying the MDE approach to the deployment aspect of a distributed ledger solution.

III. DEPLOYMENT VIEW OF DISTRIBUTED LEDGER

There are many other architectural views models apart from Kruchten's one. Rozanski and Woods [38] give examples of architectural views models, e.g.: 4 + 1, RM-ODP, Siemens, SEI. But, there is no architectural description adapted to blockchain solutions. The *Architectural views model* 1 + 5 was designed to model collaborating systems in the context of business processes [39] (see Fig. 1).

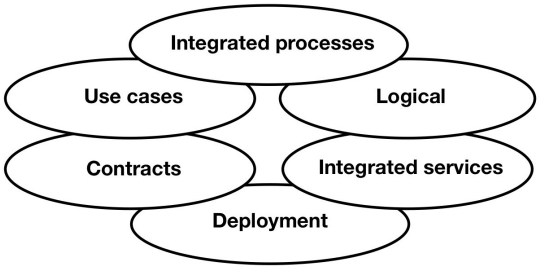

FIGURE 1. Architectural views model 1+5.

The purpose of *Integrated processes view* is the identification of business processes that should be supported by software systems. The *Use cases view* defines functions that those software systems should realize. The *Logical view* presents realizations of the use cases identified in the *Use case* view. The *Contracts view* presents contracts imposed on collaborating parties. The *Integrated services view* concentrates on communication between service providers and service consumers. The *Deployment view* defines the physical runtime environment for the solution. A broader description of those architectural views and the use of this model for designing software systems in Service-Oriented Architecture was presented by Górski [40].

As far as distributed ledger and blockchain solutions are concerned the 1 + 5 model fits perfectly. In distributed ledger solution we have collaborating parties (e.g.: seller and buyer) that act on the basis of rules defined in a smart contract. So, we have previously proposed modeling elements for representing the collaboration of parties through smart contracts [36]. We have placed those elements within the 1+5 model in new and unique, in the context of smart contracts, view – *Contracts View*. For description of smart contracts we have proposed dedicated Unified Modeling Language stereotypes, i.e.: ≪State≫, ≪Flow≫, ≪Contract≫, and ≪VerificationRule≫. We have included them in the *UML Profile for Smart Contracts*. We have proposed a flexible approach for designing smart contracts with verification rules as a *Smart Contract Design Pattern*. The UML class diagram presents classes and interfaces that constitute the *Smart Contract Design Pattern* (see Fig. 2).

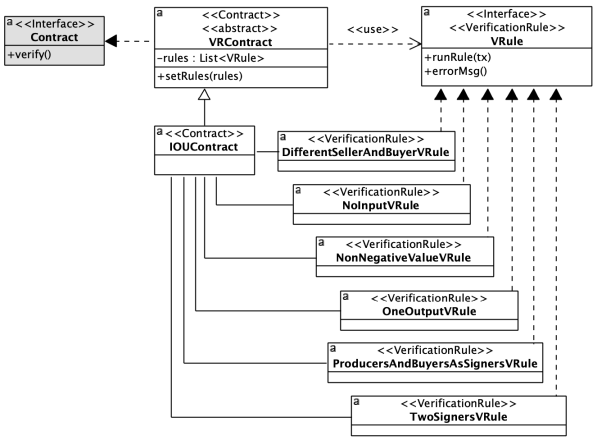

FIGURE 2. Classes and interfaces of Smart Contract Design Pattern.

A smart contract's logic is realized by a distributed application that run on multiple nodes simultaneously. The nodes, in the same network, communicate with each other in an effort to complete a specific transaction. The *Deployment view* describes a runtime environment that hosts artifacts of distributed ledger solution development process. For that reason, the *Deployment view* of the 1 + 5 model is crucial for the whole distributed ledger solution. Thus, there is a need to model the deployment environment of distributed ledger solutions and automate the creation of deployment scripts.

IV. R3 CORDA FRAMEWORK

The R3 Corda is an authenticated peer-to-peer network of nodes where each node is a Java Virtual Machine run-time environment. Each node hosts Corda services and executes

applications known as Corda Distributed Applications. We can distinguish the following node types: Network Map, Notary, Oracle, and DLTNode. Fig. 3 presents a fragment of the distributed ledger network for the ECSM system.

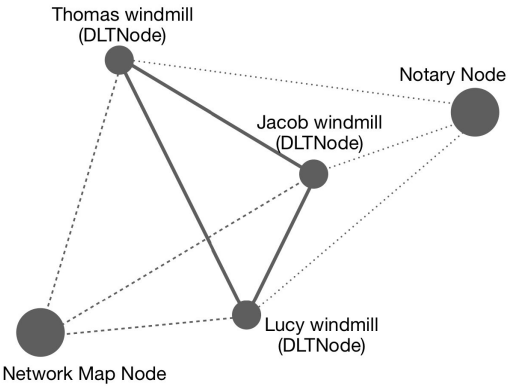

FIGURE 3. Example of a Corda network graph of connections.

Moreover, the following services are an integral part of the framework: permissioning, network map, notary, oracle, identity, and support. Using nodes and services we can constitute a Corda network, which is a fully connected graph. The graph edges do not represent persistent connections but the potential for communication. Network Map Node publishes a list of peers while Notary Node hosts notary service that provides transaction ordering and time stamping. The R3 Corda framework uses Advanced Message Queuing Protocol over Transport Layer Security (AMQP/TLS) among DLT nodes. Corda is a permissioned network. All participants of the network must use public-key infrastructure to have verifiable identities. Corda does not broadcast each transaction to the network. Only directly involved nodes are aware of transactions. Distributed ledgers are systems that enable parties who do not fully trust each other to form and maintain a consensus about a set of shared facts. The ledger from each node's point of view is the union of all intersections with other network nodes. Each network node maintains a separate vault of facts. For example, the figure shows three nodes with stored facts (see Fig. 4).

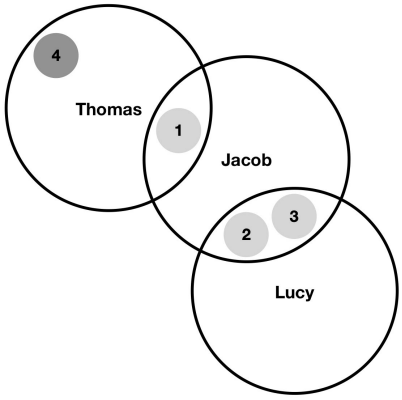

FIGURE 4. Distributed ledger facts.

Those nodes have facts in their vaults, i.e.: Thomas = {1, 4}, Jacob = {1, 2, 3}, Lucy = {2, 3}. All facts from 1 to 4 constitute facts in the distributed ledger. Only facts from 1 to 3 are shared between nodes. Corda transactions operate using consumable states which are analogous to the latest entry of a blockchain ledger that can be used to validate a new transaction.

A significant number of comparative analysis concerns the Distributed Ledger Technology frameworks. In one of the recent analysis, Chowdhury *et al.* [21] evaluate the feasibility of the most well-established, both private and public DLT platforms, i.e.: Bitcoin, Ethereum, Hyperledger Fabric, Hyperledger Sawtooth, Hyperledger Burrow, EOS, Multichain, Cardano, IOTA, Walton-Chain, and R3 Corda. They have selected a broad range of quantitative evaluation criteria. They have found that Corda consumes almost negligible energy. Secondly, Corda is a private (permissioned) DLT framework, which ensures data privacy stored. It means that the consensus in Corda is reached at the transaction level by involving relevant nodes only. It has a tremendous impact on scalability because transactions engage only two DLT and one notary node (in some cases oracle node may be also be needed). Scalability is strictly connected with performance. They have also shown that Corda is among frameworks with the shortest *block creation time* (B_{ct}, 0.5-2.0 seconds). In addition, the *block size* (B_s) is configurable. One of the basic measures of distributed ledger performance is the number of Transactions completing Per Second (*TPS*). The value of *TPS* can be calculated according to the formula (1), where TX_s is the transaction size [3].

$$TPS = \frac{B_s}{TX_s \times B_{ct}}. \tag{1}$$

With the growth of (configurable) B_s value and having constant value of TX_s we can increase *TPS* value. The shortest *block creation time* for the framework results from a consensus mechanism that engages only two DLT nodes. For example, the Corda network with $B_s = 1.0$ [MB], $TX_s = 100.0$ [B], and $B_{ct} = 0.5$ [s], can process 20,000.0 transactions per second. For comparison, the Bitcoin network can process from 7.0 to 27.0 transactions per second [3]. Moreover, the field experience has shown that Corda has broad applicability. Having all advantages in mind, we have decided to use software engineering techniques to support the deployment of distributed ledger solutions using the R3 Corda framework.

V. UML PROFILE FOR DLT DEPLOYMENT

Unified Modeling Language uses the following extensibility mechanisms: stereotypes, tagged values, and constraints [6]. These three constructs provide the ability to customize UML diagrams for a specific need. The UML specification defines stereotype as a new type of modeling element that extends the semantics of the metamodel. Stereotypes must be based on certain existing types or classes in the metamodel. Stereotypes may extend the semantics, but not the structure of pre-existing types and classes. Notation of

stereotype encloses the name in guillemets (French quotation marks), e.g., ≪include≫ or ≪Boundary≫. As far as tagged value is concerned, the UML specification states the explicit definition of a property as a name-value pair. In a tagged value, the name is referred to as the tag. The purpose of a tagged value is to assign a parameter to a model element. This enables you to enhance the description of a model element while still adhering to the UML metamodel. Tagged values must not alter or contradict the existing definition of a metaclass. Tagged values are expressed in the form 'name = value', e.g.: *author* = "Tom", *maxLoad* = 256. Constraints define rules to protect the integrity of a diagram element. The goal of a profile is to facilitate modeling in a domain by defining a set of common concepts and constructs. Profiles gather extension mechanisms to adjust UML notation.

We have used UML extensibility mechanisms for defining a profile that can be applied to the UML Deployment model of a distributed ledger solution. All proposed stereotypes and tagged values describe the needed semantic UML enrichment for the *Deployment view*. We have placed them in the *UML Profile for Distributed Ledger Deployment*.

A. STEREOTYPES IN THE PROFILE

We have used stereotypes to represent nodes, services, and communication protocols characteristic for a Corda network. We have defined the following stereotypes for nodes:

- ≪NetworkMapNode≫ — node that runs the network map,
- ≪NotaryNode≫ — node that signs transactions if their input states are valid,
- ≪OracleNode≫ — node that links the ledger to the outside world by providing facts that affect the validity of transactions,
- ≪DLTNode≫ — node that has a vault and may communicate with other nodes,
- ≪CordaNode≫ — an abstract node that gathers common properties for all types of nodes.

A Corda network comprises nodes that communicate using protocols to create and validate transactions. Nodes host and run services. We have identified the following UML stereotypes for services:

- ≪permissioningService≫ — service used to provision TLS certificates,
- ≪networkMapService≫ — service that enforces rules regarding the information that nodes must provide before being admitted to the network,
- ≪notaryService≫ — service, which is used to provide transaction ordering and time stamping,
- ≪oracleService≫ — service, which signs transactions if they state a fact, and that fact is considered to be true,
- ≪identityService≫ — service that controls admissions of participants into Corda Network,
- ≪supportService≫ — service that resolves inquiries and incidents relating to the identity and notary services.

Fig. 5 shows stereotypes declared for services in the UML Profile diagram.

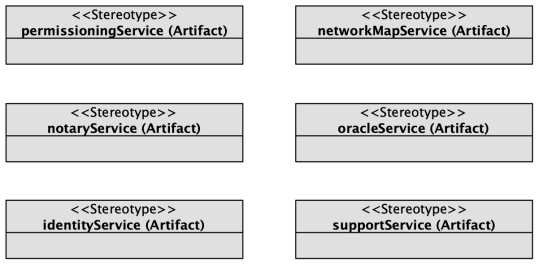

FIGURE 5. UML Profile diagram with stereotypes for DLT services.

In order to mark communication protocols we have added the following stereotypes for connection links:

- ≪HTTPS≫ — Hypertext Transfer Protocol encrypted using Transport Layer Security (HTTPS),
- ≪AMQP/TLS≫ — Advanced Message Queuing Protocol encrypted using Transport Layer Security.

Table 1 contains the summary of proposed stereotypes with extended UML elements.

TABLE 1. The summary of stereotypes in the profile.

Stereotype	Extended UML element
≪CordaNode≫	Node
≪DLTNode≫	Node
≪OracleNode≫	Node
≪NotaryNode≫	Node
≪NetworkMapNode≫	Node
≪permissioningService≫	Artifact
≪networkMapService≫	Artifact
≪notaryService≫	Artifact
≪oracleService≫	Artifact
≪identityService≫	Artifact
≪supportService≫	Artifact
≪HTTPS≫	Generic Connection
≪AMQP/TLS≫	Generic Connection

B. TAGGED VALUES

Deployment parameters can be set up for a single node configuration. Each parameter has its name and a value. In order to model deployment parameters, we have used tagged values. The full list contains 52 tagged values that represent the configuration parameters of Corda nodes. During declaring tagged values, we have intentionally omitted those deployment parameters of R3 Corda 4.3 version which are marked as deprecated, internal options, or unsupported configuration. Complete documentation of R3 Corda v.4.3 can be found at the enclosed link [20]. Starting from UML 2.0, tagged values are considered to be attributes of a stereotype. We have applied tagged values to identified stereotypes for distributed ledger nodes. In order to properly allocate tagged values to the node's stereotype, we have used an inheritance relationship from the object-oriented approach. First, we have identified tagged values that describe deployment configuration common for all types of nodes. That set of tagged values we have

placed in the ≪CordaNode≫ stereotype. Table 2 contains selected tagged values that describe the properties of all types of distributed ledger nodes.

TABLE 2. Selected tagged values for ≪CordaNode≫ stereotype.

Tagged value name	Type	Default value
additionalP2PAddresses	Text	Not defined
attachmentCacheBound	Integer	1024
custom.jvmArgs	Text	Not defined
database.initialiseSchema	Boolean	true
detectPublicIp	Boolean	false
flowMonitorPeriodMillis	Integer	60
flowTimeout.timeout	Integer	30
flowTimeout.maxRestartCount	Integer	6
h2Settings	Text	NULL
jarDirs	Text	Not defined
jmxMonitoringHttpPort	Integer	Not defined
jmxReporterType	Text	JOLOKIA
keyStorePassword	Text	cordacadevpass
messagingServerAddress	Text	Not defined
messagingServerExternal	Text	Not defined
myLegalName	Text	Not defined
networkServices.doormanURL	Text	Not defined
networkServices.pnm	Text	Not defined
p2pAddress	Text	Not defined
rpcSettings.useSsl	Boolean	false
systemProperties	Text	Not defined
trustStorePassword	Text	trustpass

The rest stereotypes for nodes inherit from the ≪CordaNode≫. Thanks to that all stereotypes for DLT nodes have the same basic set of tagged values. There is an additional set of tagged values for the ≪NotaryNode≫ which is available only for nodes running notary service.

Table 3 contains tagged values characteristic for the ≪NotaryNode≫ stereotype.

TABLE 3. Tagged values for ≪NotaryNode≫ stereotype.

Tagged value name	Type	Default value
notary.validating	Boolean	false
notary.serviceLegalName	Text	Not defined
notary.dftSMaRt.clusterAddresses	Text	Not def.
notary.raft.clusterAddresses	Text	Not def.
notary.raft.nodeAddress	Text	Not defined
notary.bftSMaRt.replicaId	Text	Not defined

Finally, we have assigned tagged values to previously declared stereotypes for distributed ledger nodes. Tagged values, common to all stereotypes, we have assigned to the ≪CordaNode≫ generic stereotype. Then we have used a generalization relationship and connected this generic stereotype with each of the DLT stereotypes. As a result, each of these stereotypes inherits a common subset of tagged values. This subset we defined once for the ≪CordaNode≫ stereotype and is available in all others. In addition, we have assigned to the ≪NotaryNode≫ stereotype a subset of tagged values characteristic only for this stereotype. The UML profile diagram presents the inheritance tree of stereotypes with tagged values (see Fig. 6).

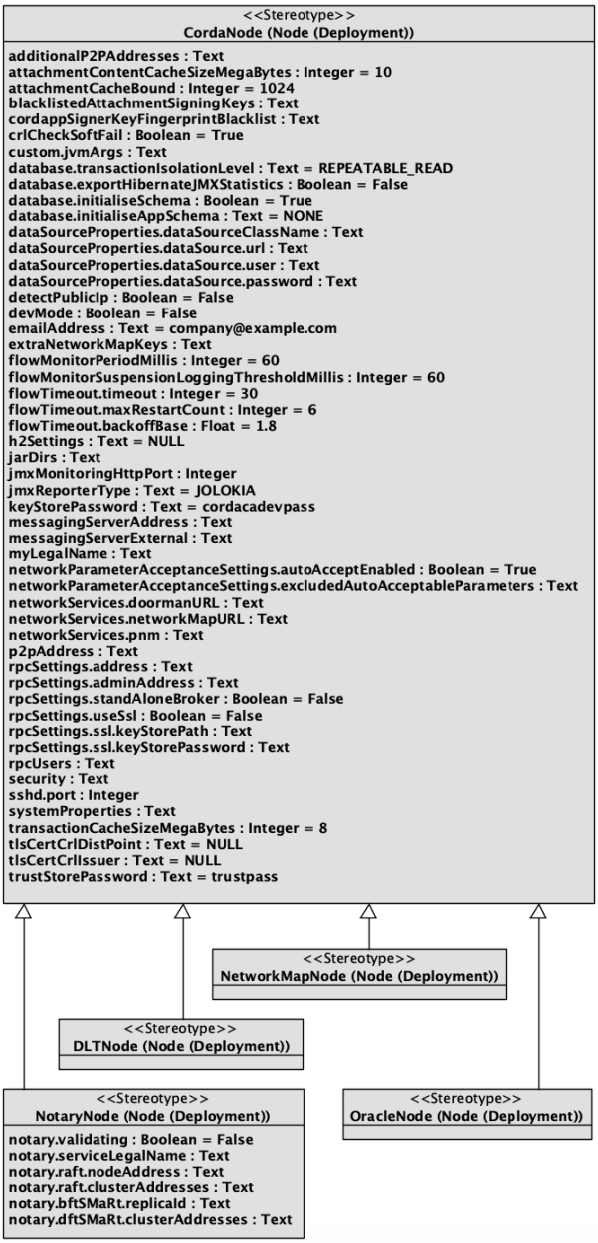

FIGURE 6. UML Profile diagram presents stereotypes for nodes with tagged values.

We have modeled the profile in Visual Paradigm Enterprise. We have placed the designed profile in the GitHub repository [41].

VI. UML2Deployment TRANSFORMATION DESIGN

Modular design is a desirable property of a transformation. Despite the fact that language support for modularisation of transformations is emerging, their designs in many cases are monolithic and contain a huge number of rules. The subject is vital and attracts researchers to tackle that

problem. For example, Fleck *et al.* [27] have proposed an automated search-based approach to modularise transformations based on higher-order transformations. Their idea was to improve the maintainability of the model transformation program.

Having that in mind, we have applied several architectural principles for the design of our transformation. First of all, *modularity* caused that we have divided our solution into the following modules: *UML Profile for Distributed Ledger Deployment*, UML Deployment model, Object-oriented model, Configuration files templates, and Configuration files. We present the overview of our MDE solution to automate the R3 Corda distributed ledger deployment (see Fig. 7).

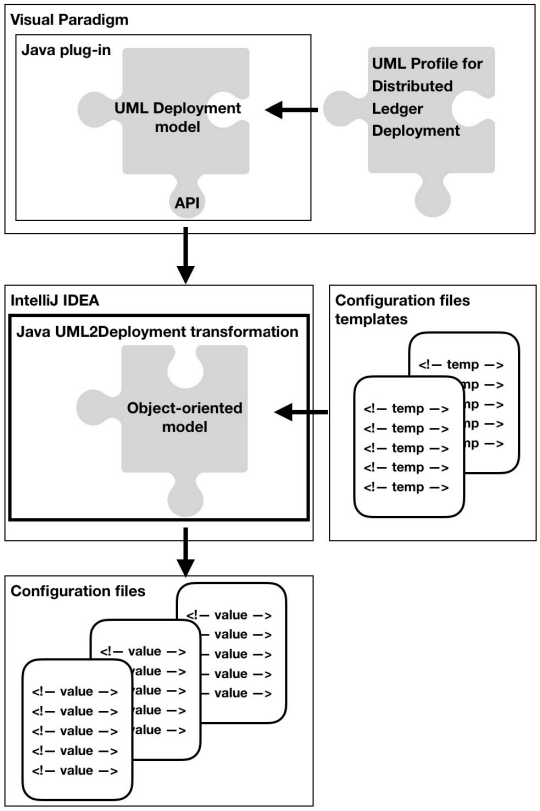

FIGURE 7. The MDE solution overview.

As far as *separation of responsibility* is concerned we have designed the transformation as two separate Java projects: transformation application that transforms a UML Deployment model into deployment configuration scripts and plug-in application that integrates the transformation with Visual Paradigm modeling tool. At a lower level that principle involves the clear division of provided functions among identified modules.

We have also applied *DLT version configurability* architectural principle. In order to achieve the possibility of changing the R3 Corda version without the need of changing the transformation we have used two following elements: configuration files templates and *UML Profile for Distributed Ledger Deployment*. Both elements use the same set of stereotypes and tagged values. That set complies with the specification of the current 4.3 version of the R3 Corda framework. The structure of the generated configuration file is configurable. The transformation fills only matching elements in files. The transformation is designed in that way that we need to provide only the new version of the profile with corresponding configuration files templates to generate configuration files for the new version of the framework. It gives us an option to manage configuration between consecutive versions of the platform also when the structure of configuration files changes. From the technical point of view, the template mechanism ensures independence from a distributed ledger platform provider. Currently, there are no standards in this area and platforms providers are managing the configuration in distinct ways. So, proposed template mechanism may be not enough to switch between various distributed ledger platforms.

In addition, our goal was also to keep independence from the modeling tool. Thus, we have applied the *interchangeability of modeling tools* architectural principle. We have introduced that architectural principle because we would like to keep the possibility to integrate easily with other modeling tools (e.g., Eclipse Papyrus). We have chosen Visual Paradigm because it has a wide spectrum of features:

- versions are available at various platforms, e.g.: macOS, Linux, Windows, Unix,
- free Community Edition is available,
- Enterprise Edition offers full support for UML extensibility and MDE,
- exposes API which allows on full access to UML models from Java source code level.

Moreover, we have a commercial license available for Visual Paradigm Enterprise at our university. It is worth emphasizing, that our approach resolves the issue of vendor lock-in.

And the last architectural principle is *simplicity*. As a result, we have identified a few modules with simple rules of transition between them. The source of the transformation is the *UML Deployment model* of the distributed ledger solution with the *UML Profile for Distributed Ledger Deployment* applied. The *UML Deployment model* may be stored in various formats depending on the modeling tool. So, we have used the Application Programming Interface (API) of the Visual Paradigm modeling tool to get the complete set of nodes with specified tagged values. That set is stored in the *Object-oriented model*. Then the *Java UML2Deployment transformation* application reads the proper *Configuration files templates* and generates deployment *Configuration files*. The transformation generates a Gradle Groovy DSL file (*deployNodesTask.gradle*). The file contains the fulfilled Gradle task with the required deployment configuration parameters of the distributed ledger network. For each node, the transformation also generates a file with the complete set of deployment parameters.

Next, we have presented elements of our transformation, i.e.: the UML deployment model, the Object-oriented model, configuration files templates, and configuration files. We have also described the transformation algorithm.

A. THE UML DEPLOYMENT MODEL

The definition of the UML Deployment model requires providing meta-model syntax and semantics [33]. A meta-model (M2) is a model that describes the artifacts of and rules for a model, e.g., classes, nodes, stereotypes, and tagged values. A model (M1) is the model that describes the artifacts and rules for the problem domain. At that level, we draw UML diagrams. We can depict the UML Deployment model into two types of UML diagrams: component and deployment. The first one represents physical pieces of software used by the implemented system. The latter shows the deployment environment of the designed system. Together the component and deployment diagrams describe how a software system is installed on the runtime environment. Fig. 8 shows a fragment of the mapping between the M2 meta-model and the M1 model for our Deployment model.

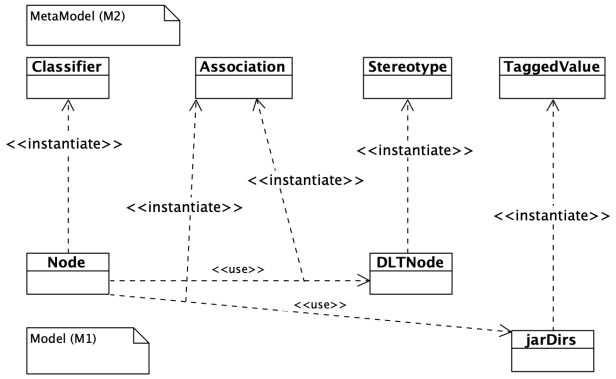

FIGURE 8. The mapping between the M2 meta-model and the M1 model.

The first module is the UML Deployment model. At that level, network nodes are represented by UML Nodes objects with configuration attributes defined as tagged values. Each node has a proper stereotype and predefined list of tagged values with default values. Stereotypes and tagged values definitions are provided by the *UML Profile for Distributed Ledger Deployment*. Packages group nodes. The package represents one deployment environment, e.g.: dev, test, uat, pre-prod, prod. The source of the transformation is one of the deployment environments in the UML Deployment model of the DLT solution with the *UML Profile for Distributed Ledger Deployment* applied. Thus, the source of the transformation comprises nodes, services, and communication links. We can use a single UML Deployment diagram or many of them to describe the UML Deployment model of the DLT solution. Fig. 9 presents an example of the UML Deployment diagram for a fragment of the ECSM system [4], [36].

Visual Paradigm tool stores models in binary flat files (with.vpp extension). Other tools can not read them without understanding the tool's internal format. We identified two ways to extract needed data in the format which we can use in our transformation. The first one relies on the UML model export of to the XML file, which shows data in a readable format. The second approach, selected by us, relies

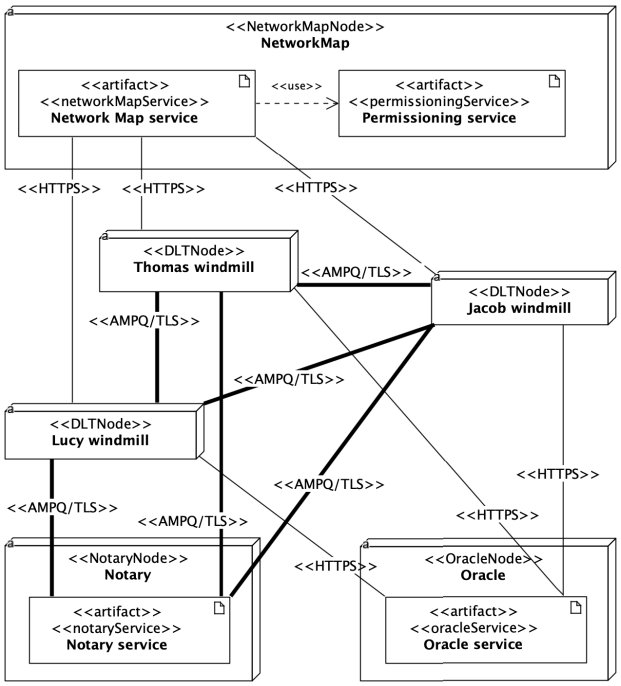

FIGURE 9. The UML Deployment diagram for the ECSM system.

on the usage of the delivered API. It allows for reading and manipulating data contained inside the UML Deployment model. That method allows us to read data directly into the source code of the Object-oriented model and speed up the process of data extraction.

Table 4 shows the mapping between the UML Deployment model and Visual Paradigm API elements.

TABLE 4. The mapping between the UML Deployment model and Visual Paradigm API modeling elements.

UML element	API element
Package	com.vp.plugin.model.IPackage
Node	com.vp.plugin.model.INode
TaggedValue	com.vp.plugin.model.ITaggedValue

We have designed the transformation for the generation of deployment configuration at two levels:

- deployment environment level — we generate deployment configuration for a UML package which represents deployment environment,
- node level — we generate deployment configuration for a single node.

In case of deployment environment level, the transformation gathers all objects that implement *INode* interface, which are children of the *IPackage* interface. Fig. 10 shows the source code of the *getNodes* method that gathers UML nodes and creates collection of objects that implement *INode* interface.

In the case of node level, the transformation reads only a single object implementing *INode* interface that represents the UML node.

```
public Collection<INode> getNodes(IPackage vpPackage) {
 UIHelper.logMessage("Collecting nodes details...");

 Iterable<INode> nodesIterable = () ->
   vpPackage.childIterator(IModelElementFactory.MODEL_TYPE_NODE);

 List<INode> nodes = StreamSupport
   .stream(nodesIterable.spliterator(), false)
   .collect(Collectors.toList());

 UIHelper.logMessage("Nodes search finished
   (" + nodes.size() + " nodes found)");

 return nodes;
}
```

FIGURE 10. The source code of the *getNodes* method.

B. THE OBJECT-ORIENTED MODEL

The Object-oriented model (OOM) comprises Java classes and interfaces that directly correspond to classes from the Visual Paradigm API and indirectly to UML nodes. We have added the interface *CordaObject* and the class *CordaNode* to provide a higher level of abstraction for all objects created on the basis of the UML Deployment model. From the Java source code perspective, such an approach allows us on designing and developing the transformation algorithm in a generic and extendable way. The actual object class is hidden under the parent class or interface.

Fig. 11 presents interfaces and classes in the inheritance tree of the Object-oriented model.

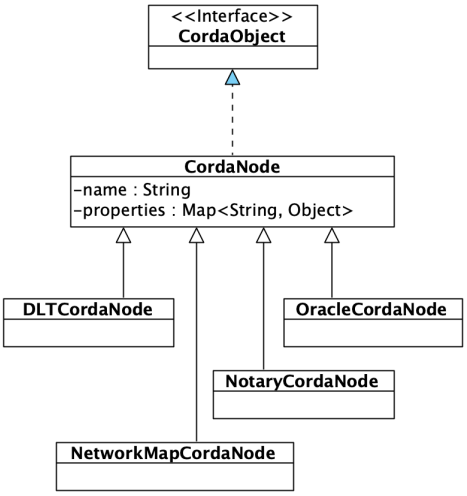

FIGURE 11. The Object-oriented model reference types.

Using the tool's API we have loosely coupled the UML Deployment model and the OOM. Table 5 shows the mapping between the Visual Paradigm API and the OOM classes.

TABLE 5. The mapping between the tool API and the Object-oriented model.

API element	Object-oriented model class
INode with ≪DLTNode≫	DLTCordaNode
INode with ≪NotaryNode≫	NotaryCordaNode
INode with ≪OracleNode≫	OracleCordaNode
INode with ≪NetworkMapNode≫	NetworkMapCordaNode

The transformation creates Java objects, in the OOM, based on the collection of objects that implement the *INode* interface. Each Java object has its reference type. The type depends on the stereotype applied to the *INode* interface. We can read the stereotype by using the *hasStereotype* method provided by *INode* interface. The API allows us to manipulate the UML Deployment model with the usage of an object-oriented approach where UML elements are reflected inside the source code as interfaces, classes, and objects. At the Java source code level, the creation of proper objects is done with the usage of the *StereotypesEnum* data type (see Fig. 12).

```
public enum StereotypesEnum {

    DLTNode, OracleNode, NetworkMapNode, NotaryNode;

    public CordaNode getInstance() {
        switch (this) {
            case NotaryNode:
                return new NotaryCordaNode();
            case OracleNode:
                return new OracleCordaNode();
            case NetworkMapNode:
                return new NetworkMapCordaNode();
            case DLTNode:
                return new DLTCordaNode();
            default:
                return new CordaNode();
        }
    }
}
```

FIGURE 12. *StereotypesEnum* enum data type.

We have applied inheritance and polymorphism to store objects of all that classes in one generic list of type List<CordaNode>. Moreover, we have also applied Java generic HashMap type to store tagged values in Java classes, Map<String, Object>. The values of tags can be read by using the API. The *INode* interface has convenient methods for that purpose. Fig. 13 presents the source code of the *readTaggedValue* method.

```
private Object readTaggedValue(String key, INode vpNode) {
   ITaggedValue vpTaggedValue =
       vpNode.getTaggedValues().getTaggedValueByName(key);
   if (vpTaggedValue == null) {
     return null;
   }
     return vpTaggedValue.getValue();
}
```

FIGURE 13. The source code of the *readTaggedValue* method.

We retrieve the actual list of all tagged values from the *UML Profile for Distributed Ledger Deployment* using the tool API. The last activity is setting proper values for properties of objects from the List<CordaNode> corresponding to tagged values read from *INode* interfaces.

C. CONFIGURATION FILES TEMPLATES

Templates are flat files that have the same structure as configuration files. The difference lays in placeholders. We have used them to distinguish the value part of the tagged value. Placeholder for value has the same name as the tagged value

in the UML Deployment model wrapped by '<!–' prefix and '-->' suffix. Table 6 contains examples of mapping between UML tagged values and template placeholders.

TABLE 6. Mapping between UML tagged values and template placeholders.

UML tagged value	Template placeholder
myLegalName	<!-- myLegalName -->
p2pAddress	<!-- p2pAddress -->
custom.jvmArgs	<!-- custom.jvmArgs -->
rpcSettings.useSsl	<!-- rpcSettings.useSsl -->

Fig. 14 presents a fragment of the *nodeConfig.template* configuration file template.

```
myLegalName = "<!--myLegalName-->"
additionalP2PAddresses =
     <!--additionalP2PAddresses-->
attachmentContentCacheSizeMegaBytes =
     <!--attachmentContentCacheSizeMegaBytes-->
attachmentCacheBound = <!--attachmentCacheBound-->
blacklistedAttachmentSigningKeys =
     <!--blacklistedAttachmentSigningKeys-->
cordappSignerKeyFingerprintBlacklist =
     <!--cordappSignerKeyFingerprintBlacklist-->
crlCheckSoftFail = <!--crlCheckSoftFail-->
custom = {
    jvmArgs = <!--custom.jvmArgs-->
}
database = {
    transactionIsolationLevel =
     <!--database.transactionIsolationLevel-->
    exportHibernateJMXStatistics =
     <!--database.exportHibernateJMXStatistics-->
}
```

FIGURE 14. Fragment of configuration file template.

The transformation selects proper configuration template files for objects from the List<CordaNode>. Table 7 shows template files corresponding to the OOM elements.

TABLE 7. Mapping between the Object-oriented model and template files.

OOM element	Template file
List<CordaNode>	deplyNodesTask.template
DLTCordaNode	nodeConfig.template deployNodesTask_node.template
NetworkMapCordaNode	nodeConfig.template deployNodesTask_node.template
OracleCordaNode	nodeConfig.template deployNodesTask_node.template
NotaryCordaNode	notaryNodeConfig.template deployNodesTask_notaryNode.template

D. CONFIGURATION FILES

Gradle is a general-purpose build management system [42]. Gradle supports the automatic download and configuration of dependencies or other libraries. It supports Maven repositories for retrieving these dependencies. Gradle builds are described via one or multiple *build.gradle* files. At least one *build.gradle* file is typically located in the root folder of the project. Each of these files defines a project and its tasks. In Gradle, we have projects that mean something to build or work to do. Each project comprises tasks. A task represents a piece of work that a build performs, e.g., compile the source code or generate the Javadoc. These build files are based on a Groovy Domain Specific Language. In this file, you can use a combination of declarative and imperative statements. You can also write Groovy or Kotlin code whenever you need it. Tasks can also be created dynamically at runtime. The *build.gradle* file comprises three types of elements:

- task — a *task* represents a single atomic piece of work for a build, such as compiling classes. We have incorporated the *deployNodes* task into generation.
- block — a build script is made up of zero or more statements and script blocks. Statements can include method calls, property assignments, and local variable definitions. We have used the *node* block for node configuration in the transformation.
- property — in the *node* block, we set values of properties based on corresponding values of tagged values.

Gradle uses the following files to configure and deploy a distributed ledger network:

- build.gradle — expressed in Groovy DSL or Kotlin DSL. We have used Groovy DSL due to performance reasons,
- node.conf — complete set of configuration options for a specific node.

We have customized *deployNodes* task to configure and generate a set of distributed ledger nodes. That task is used by *build.gradle* to configure the network of nodes. The transformation also generates a configuration file for node. The file uses the Human-Optimized Config Object Notation (HOCON) format [43]. HOCON format is a superset of JavaScript Object Notation (JSON) [44].

The transformation generates configuration files by populating the content of each file with templates. It replaces placeholders with proper tagged values (see Fig. 15).

```
private String populateTemplateWithTags(String template,
                                Map<String, Object> tagsAndValues) {
 for (Map.Entry<String, Object> entry
                                    : tagsAndValues.entrySet()) {
  String key = entry.getKey();
  Object value = entry.getValue();
  String tag = TemplateTags.getInstance().buildTemplateTag(key);
  template = template.replace(tag, getValueAsString(tag, value));
 }
 return template;
}
```

FIGURE 15. The source code of the *populateTemplateWithTags* method.

For tagged values not set in the UML Deployment model placeholders remain in the generated file.

The transformation stores configuration files, generated with *nodeConfig* and *notaryNodeConfig* templates, in separate node configuration files (e.g.: NetworkMap.config, Oracle.config, Notary.config, Jacob windmill.config, Lucy windmill.config, Thomas windmill.config).

Fig. 16 presents the fragment of generated configuration file for the *Windmill Thomas* node.

```
myLegalName = "O=Windmill Thomas,L=Gdynia,C=PL"
additionalP2PAddresses =
   <!--additionalP2PAddresses-->
attachmentContentCacheSizeMegaBytes = 10
attachmentCacheBound = 1024
blacklistedAttachmentSigningKeys =
   <!--blacklistedAttachmentSigningKeys-->
cordappSignerKeyFingerprintBlacklist =
   <!--cordappSignerKeyFingerprintBlacklist-->
crlCheckSoftFail = True
custom = {
    jvmArgs = -Xms256m -Xmx768m
}
database = {
    transactionIsolationLevel = REPEATABLE_READ
    exportHibernateJMXStatistics = False
}
```

FIGURE 16. The fragment of the generated configuration file.

All configuration files generated with *deployNodeTask* types templates are combined into one file. The content prepared in that way is injected into *deplyNodesTask.template*. The transformation uses that template and creates *deployNodesTask.gradle* configuration file.

E. THE TRANSFORMATION ALGORITHM

There are two entry points of the transformation algorithm: all nodes, from the selected environment, configuration generation, and a single node configuration generation. Thus, we have enumerated initial steps of the algorithm as *1a* and *1b*, respectively. Similarly, the Step 3a applies to the generation of deployment configuration for the selected UML package and the Step 3b refers to the generation of a single node configuration. The transformation acts according to the following algorithm:

- **Step 1a.** A user invokes *Generate Nodes configuration* option to generate the configuration of all nodes for the selected UML package.
- **Step 1b.** A user invokes *Generate Node configuration* option to generate the configuration for the single node,
- **Step 2.** A user selects destination folder for generated configuration files.
- **Step 3a.** The transformation gathers all UML nodes, which are places in the selected package, and builds a list of objects that implement the *INode* interface.
- **Step 3b.** The transformation selects the UML node and creates a list that contains one object, which implements *INode* interface.
- **Step 4.** For each object on the list, the transformation creates a Java object of the proper class, based on the applied stereotype, e.g., an object of DLTCordaNode class for ≪DLTNode≫ stereotype. The transformation creates a collection of objects, Collection<CordaNode>.
- **Step 5.** The transformation reads template file for the Gradle *deployNodes* task.
- **Step 6.** For each Java object from the collection, the transformation:
  - *Step 6.1.* reads template for the Gradle *node* block,
  - *Step 6.2.* populates values in the Gradle *node* block template with values stored in the Java object,
  - *Step 6.3.* adds generated Gradle *node* block to the Gradle *deployNodes* task template,
  - *Step 6.4.* reads template for node configuration file,
  - *Step 6.5.* populates values in node configuration file template with values stored in the Java object,
  - *Step 6.6.* saves the single node configuration file to the selected destination.
- **Step 7.** The transformation saves the Gradle *deployNodes* task to the proper destination.
- **Step 8.** The transformation informs the user about the finished generation.

We have designed the transformation in IntelliJ IDEA. We have used the Lombok library, which is a fully-featured builder [46]. By using annotations of the library we were able to automatically generate class methods, e.g.: a getter, setter, equals, or constructor. That allowed us to limit the program code to statements related to business logic. As a result, the source code is shorter and clearer. We have placed the designed transformation application in the GitHub repository [47]. Furthermore, we have designed the Java plug-in and integrated the transformation with Visual Paradigm Enterprise. We have placed the designed Java plug-in application in the GitHub repository [48].

VII. THE TRANSFORMATION USAGE

We have designed two options to run the transformation in the Visual Paradigm modeling tool:

- generation of single-node deployment configuration - we have added the option *Generate Node configuration* to the UML Node context menu in the UML Deployment diagram or Model Explorer,
- generation of deployment configuration for a set of nodes declared in the UML Package - we have added the option *Generate Nodes configuration* to the UML Package context menu in Model Explorer.

Thus, the transformation can be run at two levels. At the node level, the transformation generates deployment configuration for a single node. At the deployment environment level, the transformation generates deployment configuration for a UML package which represents deployment environment, e.g.: dev, test, uat, pre-prod, prod.

Fig. 17 shows the context menu of the Visual Paradigm tool extended with the *Generate Node configuration* option.

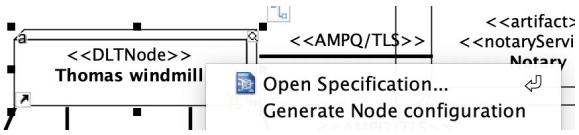

FIGURE 17. Extended context menu of the UML node.

We can run the transformation for a single node *Thomas windmill* marked with ≪DLTNode≫ stereotype. As a result, we will get two files: *deployNodesTask.gradle* and *Thomas windmill.config*. We present a fragment of generated script file (*deployNodesTask.gradle*) for configuration of *Thomas windmill* distributed ledger node (see Fig. 18).

```
node {
  name "O=Thomas windmill,L=Gdynia,C=PL"
  devMode false
  p2pAddress "thomas.corda.amw.gdynia.pl:10002"
  rpcSettings {
    useSsl false
    standAloneBroker false
    address "thomas.corda.amw.gdynia.pl:10003"
    adminAddress "thomas.corda.amw.gdynia.pl::10103"
  }
  rpcUsers = [[ user: "thomas", "password": "password",
   "permissions": ["ALL"]]]
   configFile = "./build/nodes/thomas/thomas.conf"
}
```

FIGURE 18. Fragment of the *deployNodesTask.gradle* script.

The second option is to generate a deployment configuration for all nodes in a specific UML package. The option is available in the *Model Explorer*. In software engineering, we usually need several environments that organize the development process, e.g.: development, tests, user acceptance tests, pre-production, and production. We present the *Model Explorer* view with three UML packages that represent deployment environments, i.e.: dev, test, and prod (see Fig. 19).

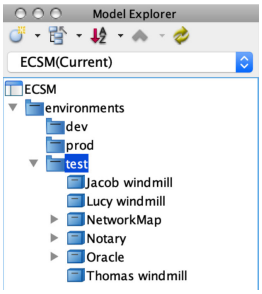

FIGURE 19. Model Explorer with several deployment environments.

A clear and repeatable deployment process is crucial for Continuous Delivery. The second option of the transformation allows for managing multiple deployment environments within the software development project. UML packages reflect deployment environments. We can include UML modeling into the Continuous Delivery process and support automatic environments building and deployment. Configuration managers responsible for managing configurations across multiple environments after introducing changes or adding new environments generate actual setup and save it in dedicated network localization. When the new deployment is in progress (e.g., with the usage of automation servers like Jenkins or Bamboo), the proper configuration can be automatically read from defined localization and included in the deployment.

VIII. THE TRANSFORMATION VALIDATION

One of the important aspects of the paper is the validation of the proper functioning of the transformation. According to IEEE 610.12-1990 Standard, validation is the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements [53]. Therefore, we have concluded that it requires comparing the source and the target of the transformation, and thus checking the consistency of both elements. We have adopted the following assumptions for validation: the source is a UML node in the UML Deployment model and the target is the deployment configuration file for that UML node.

Both those elements comprises a set of parameters. The source encompasses set M that comprises tagged values m, $m \in M$. The target contains set C, which consists of deployment configuration parameters c, $c \in C$ (see Fig. 20).

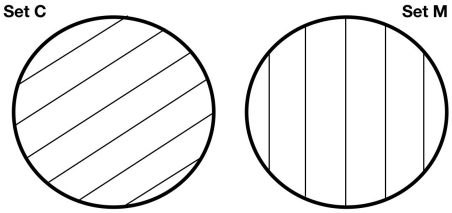

FIGURE 20. Two separate sets of parameters to validate.

Intersection of two sets C and M is denoted by $C \cap M$, and is the set containing all elements of C that also belong to M (or equivalently, all elements of M that also belong to C). We have to check that intersection of those two sets satisfies the following equation (see formula (2)).

$$C \cap M = C = M. \tag{2}$$

To do that, we have to verify whether those two sets contain the same elements with the same values (see Fig. 21).

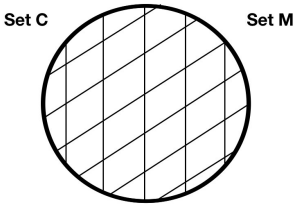

FIGURE 21. The intersection of identical sets *C* and *M*.

Conditions for checking intersection $C \cap M$ encompass the following formulas. The cardinality of both sets should be the same (see formula (3)).

$$|C| = |M|. \tag{3}$$

We should define how to compare elements in both sets. The deployment configuration parameter c is an ordered pair, $c = (n^c, v^c)$, where:

- n^c – name of the deployment configuration parameter c,
- v^c – value of the deployment configuration parameter c.

The tagged value m is an ordered pair, $m = (n^m, v^m)$, where:

- n^m – name of the tagged value m,
- v^m – value of the tagged value m.

For each deployment configuration parameter $c \in C$ there should be corresponding tagged value $m \in M$ which has the same name and value (see formula 4).

$$\bigwedge_{c \in C} \bigvee_{m \in M} (n^c = n^m) \wedge (v^c = v^m). \tag{4}$$

The same must be fulfilled from the opposite side. For each tagged value $m \in M$ there should be corresponding deployment configuration parameter $c \in C$ which has the same name and value (see formula 5).

$$\bigwedge_{m \in M} \bigvee_{c \in C} (n^m = n^c) \wedge (v^m = v^c). \tag{5}$$

We also need to check the relative complement of C in M, denoted by $M \setminus C$, to verify that the set of elements in M but not in C is empty (see formula (6).

$$M \setminus C = \varnothing. \tag{6}$$

We have also done a reverse check of the relative complement of $C \setminus M$ to verify the set of elements in C but not in M is empty (see formula (7).

$$C \setminus M = \varnothing. \tag{7}$$

At the design level, we have developed a separate validation application that does not use any element of the transformation application. The Java validation application checks two *HashMap* collections. The first one contains objects corresponding to deployment configuration parameters. Whereas, the second one comprises objects that refer to tagged values from the UML node. The validation application checks the intersection of those collections in its general form (see Fig. 22).

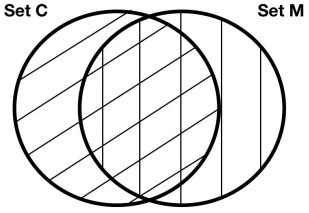

FIGURE 22. The intersection of sets *C* and *M* in general form.

Thus, the validation application looks for objects that appear in both collections, and it also searches for objects, which appear only in one of them.

We present the architectural overview of Java application *UML2DeploymentCheck* that validates distributed ledger deployment configuration files (see Fig. 23).

We have integrated the validation application with the Java plug-in application. In that way, we have expanded the UML node context menu in Visual Paradigm with the *Validate Node configuration* option. After invoking the *Validate Node configuration* option on the selected UML node, we have to choose the deployment configuration file for comparison. The validation application does the following actions:

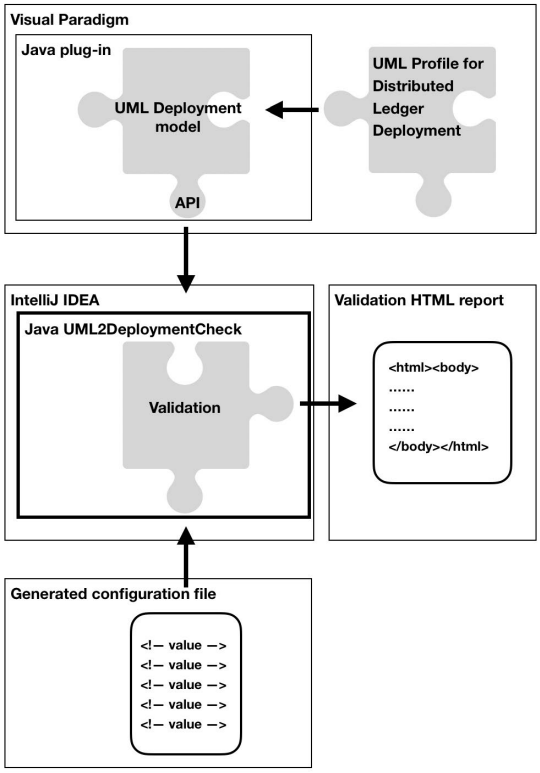

FIGURE 23. The validation application overview.

- reads all tagged values from the selected UML node and creates corresponding objects in the model *HashMap* collection,
- reads all deployment configuration parameters from the chosen deployment configuration file and creates corresponding objects in the deployment *HashMap* collection,
- for each object in the deployment *HashMap* collection, the validation application looks for the corresponding object in the model *HashMap* collection and adds the result to the HTML report,
- for each object in the model *HashMap* collection, the validation application looks for the corresponding object in the deployment *HashMap* collection and adds the result to the HTML report.

As a result of running the validation of the UML node deployment configuration, the HTML report is created and saved in the same destination where the deployment configuration file resides. In that HTML report:

- matching entries are marked in white,
- entries with differences are marked in red,
- entries found in the configuration file but not found in the UML node are marked in yellow,
- entries found in the UML node but not found in the configuration file are marked in orange.

Fig. 24 shows a fragment of HTML report from validation with deliberately made changes to emphasize differences.

No	Key	UML value	Config value
1	additionalP2PAddresses	null	null
2	attachmentCacheBound	1024	1024
3	attachmentContentCacheSizeMegaBytes	10	10
4	blacklistedAttachmentSigningKeys	null	null
5	cordappSignerKeyFingerprintBlacklist	null	null
6	crlCheckSoftFail	True	True
7	custom.jvmArgs	-Xms256m -Xmx768m	-Xms256m -Xmx768m
8	dataSourceProperties.dataSource.password	""	""
9	dataSourceProperties.dataSource.url	null	null
10	dataSourceProperties.dataSource.user	sa	sa
11	dataSourceProperties.dataSourceClassName	org.h2.jdbcx.JdbcDataSource	org.h2.jdbcx.JdbcDataSource
12	database.exportHibernateJMXStatistics	False	False
13	database.initialiseAppSchema	NONE	NONE
14	database.initialiseSchema	True	False
15	database.transactionIsolationLevel	REPEATABLE_READ	REPEATABLE_READ
16	detectPublicIp	False	False
17	devMode	False	null
18	emailAddress	thomas@amw.gdynia.pl	thomas@amw.gdynia.pl
19	extraNetworkMapKeys	null	null
20	flowMonitorPeriodMillis	60	60

FIGURE 24. Excerpt of HTML report from the validation.

As a result, we have obtained the tool to validate consistency between UML models and deployment configuration files for nodes. We have conducted tests for the ECSM model. Tests have confirmed that the transformation works correctly. We have designed and developed the validation application in IntelliJ IDEA and integrated it with Visual Paradigm Enterprise. We have placed the designed validation application in the GitHub repository [52].

IX. DISCUSSION AND LIMITATIONS

From the literature review, we can conclude that there is a very limited amount of research work done on the application of Model-Driven Engineering to distributed ledger and blockchain technologies. Most of them focus on smart contracts. We used MDE to manage the deployment configuration of the R3 Corda distributed ledger solution.

The transformation is designed in such a flexible way that allows on updating between distributed ledger platform versions without the need to change the source code. The Java transformation is loosely coupled with the distributed ledger platform by using the dedicated UML profile and deployment configuration files. Templates give the possibility to manage configuration differences between DLT platform versions, like new or deprecated deployment parameters or changes in the deployment configuration file structure. In case of new deployment parameters appearance or change of the existing ones, the following steps must be done to accommodate the consecutive version:

- update of tagged values in the *UML Profile for Distributed Ledger Deployment*,
- verification of the UML Deployment model,
- update templates of deployment configuration files.

The Java transformation remains unchanged. It is worth emphasizing, that we can do the mentioned actions at runtime with no changes in the source code.

In addition, we tried to achieve a certain level of independence from the specific distributed ledger platform. Unfortunately, due to architectural differences between the distributed ledger platforms, our solution does not provide a transition to another platform without changes in the source code of the transformation. We propose steps with must be done to change the distributed ledger platform:

- preparation of a new UML profile dedicated to the new platform,
- preparation of new configuration files templates dedicated to the new platform,
- changes in selected steps of the transformation algorithm:
  - using of new stereotypes dedicated to the new platform,
  - creating new classes in the Object-oriented model that reflect new platform's stereotypes,
  - mapping between new stereotypes and the Object-oriented model,
  - mapping between the Object-oriented model and deployment configuration templates files.

At the algorithm level, the flow of activities in our transformation remains the same. It would be worth considering the design of a platform-independent model that would ensure the portability of modeled deployment environments between DLT platforms. The deployment level is strongly associated with the distributed ledger platform. As a result, we should manage many various models: one platform-independent model and a set of platform-specific models.

We have developed the transformation in the Java programming language and integrated it with the Visual Paradigm Enterprise modeling framework by separate Java plug-in. The proposed solution makes the transformation application independent of the modeling tool. We can replace the modeling tool in our solution as long as it provides an API to read the content of the models and supports UML extension mechanisms.

It is worth emphasizing that our solution allows for managing models of different deployment environments in the UML Deployment model. Moreover, we can generate a deployment configuration of the distributed ledger network for various environments. Here is the second advantage of Java. Because selected Continuous Delivery automation servers are written in Java we can incorporate our UML modeling support into the Continuous Integration / Continuous Deployment process.

We see two additional limitations to our solution. Fist of all, our transformation is unidirectional in its current form. It can generate deployment configuration scripts for the UML Deployment model, environment, or single node. In addition, the validation application can check the consistency between the model and the configuration file. However, we cannot update the model using the transformation due to changes made to the configuration file. Secondly, we have focused mainly on the *Deployment view* of the distributed ledger solution. We consider including the *Logical* and *Contracts* views

from the *Architectural views model 1+5* to encompass smart contracts and distributed applications that realize business logic.

X. CONCLUSION AND FUTURE WORK

This article introduces an innovative approach for generating distributed ledger deployment configuration files from the UML Deployment model. We have identified modeling elements of the selected distributed ledger platform and proposed new UML notation extensions in the form of dedicated — *UML Profile for Distributed Ledger Deployment*. Furthermore, we have automated the generation of the R3 Coda distributed ledger network configuration. We have developed the transformation in Java programming language and plugged-in it into the Visual Paradigm Enterprise modeling tool. We have shown the applicability of the *Architectural views model 1+5* in the *Deployment view* part for decentralized solutions. Our approach comprises UML extension mechanisms, the Model-to-Code transformation integrated with the modeling tool, and the validation application.

Future work includes improving and extending the proposed approach to support other distributed ledger platforms. We plan to propose platform-independent UML modeling extensions and create a common higher level of abstraction for deployment modeling. Furthermore, we plan to refine and combine work done from *Logical*, *Contract* and *Deployment* views into one solution. We consider including our modeling support into the Continuous Deployment process. We plan to directly integrate our transformation application with the Jenkins open source automation server.

REFERENCES

[1] X. Xu, I. Weber, and M. Staples, *Architecture for Blockchain Applications*. Cham, Switzerland: Springer, 2019, pp. 5–7, doi: 10.1007/978-3-030-03035-3.

[2] V. Gramoli, "From blockchain consensus back to Byzantine consensus," *Future Gener. Comput. Syst.*, vol. 107, pp. 760–769, Jun. 2020, doi: 10.1016/j.future.2017.09.023.

[3] C. T. Nguyen, D. T. Hoang, D. N. Nguyen, D. Niyato, H. T. Nguyen, and E. Dutkiewicz, "Proof-of-stake consensus mechanisms for future blockchain networks: Fundamentals, applications and opportunities," *IEEE Access*, vol. 7, pp. 85727–85745, 2019, doi: 10.1109/ACCESS.2019.2925010.

[4] T. Gorski, J. Bednarski, and Z. Chaczko, "Blockchain-based renewable energy exchange management system," in *Proc. 26th Int. Conf. Syst. Eng. (ICSEng)*, Dec. 2018, pp. 1–6, doi: 10.1109/ICSENG.2018.8638165.

[5] P. B. Kruchten, "The 4+1 view model of architecture," *IEEE Softw.*, vol. 12, no. 6, pp. 42–50, 1995, doi: 10.1109/52.469759.

[6] T. Pender, *UML Bible*. Indianapolis, IN, USA: Wiley, 2003.

[7] A. J. Kleppe, J. Warmer, and W. Bast, *MDA Explained, The Model Driven Architecture: Practice and Promise*. Boston, MA, USA: Addison-Wesley, 2003.

[8] N. Yousaf, F. Azam, W. H. Butt, M. W. Anwar, and M. Rashid, "Automated model-based test case generation for Web user interfaces (WUI) from interaction flow modeling language (IFML) models," *IEEE Access*, vol. 7, pp. 67331–67354, 2019, doi: 10.1109/ACCESS.2019.2917674.

[9] T. Mens and P. Van Gorp, "A taxonomy of model transformation," *Electron. Notes Theor. Comput. Sci.*, vol. 152, pp. 125–142, Mar. 2006, doi: 10.1016/j.entcs.2005.10.021.

[10] I. Santiago, Á. Jiménez, J. M. Vara, V. De Castro, V. A. Bollati, and E. Marcos, "Model-driven engineering as a new landscape for traceability management: A systematic literature review," *Inf. Softw. Technol.*, vol. 54, no. 12, pp. 1340–1356, Dec. 2012, doi: 10.1016/j.infsof.2012.07.008.

[11] J. Al-Jaroodi and N. Mohamed, "Blockchain in industries: A survey," *IEEE Access*, vol. 7, pp. 36500–36515, 2019, doi: 10.1109/ACCESS.2019.2903554.

[12] A. A. Monrat, O. Schelen, and K. Andersson, "A survey of blockchain from the perspectives of applications, challenges, and opportunities," *IEEE Access*, vol. 7, pp. 117134–117151, 2019, doi: 10.1109/ACCESS.2019.2936094.

[13] P. Gonczol, P. Katsikouli, L. Herskind, and N. Dragoni, "Blockchain implementations and use cases for supply Chains–A survey," *IEEE Access*, vol. 8, pp. 11856–11871, 2020, doi: 10.1109/ACCESS.2020.2964880.

[14] K. Leng, Y. Bi, L. Jing, H.-C. Fu, and I. Van Nieuwenhuyse, "Research on agricultural supply chain system with double chain architecture based on blockchain technology," *Future Gener. Comput. Syst.*, vol. 86, pp. 641–649, Sep. 2018, doi: 10.1016/j.future.2018.04.061.

[15] A. Shahnaz, U. Qamar, and A. Khalid, "Using blockchain for electronic health records," *IEEE Access*, vol. 7, pp. 147782–147795, 2019, doi: 10.1109/ACCESS.2019.2946373.

[16] L. Ismail, H. Materwala, and S. Zeadally, "Lightweight blockchain for healthcare," *IEEE Access*, vol. 7, pp. 149935–149951, 2019, doi: 10.1109/ACCESS.2019.2947613.

[17] T. Górski, K. Marzantowicz, and M. Szulc, "Cloud-enabled warship's position monitoring with blockchain," in *Smart Innovations in Engineering and Technology*, R. Klempous and J. Nikodem, Eds. Cham, Switzerland: Springer, 2020, pp. 53–74, doi: 10.1007/978-3-030-32861-0_4.

[18] S. Wang, A. F. Taha, J. Wang, K. Kvaternik, and A. Hahn, "Energy crowdsourcing and peer-to-peer energy trading in blockchain-enabled smart grids," *IEEE Trans. Syst., Man, Cybern. Syst.*, vol. 49, no. 8, pp. 1612–1623, Aug. 2019, doi: 10.1109/TSMC.2019.2916565.

[19] X. Lu, Z. Guan, X. Zhou, X. Du, L. Wu, and M. Guizani, "A secure and efficient renewable energy trading scheme based on blockchain in smart grid," in *Proc. IEEE 5th Int. Conf. Data Sci. Syst. (HPCC/SmartCity/DSS)*, Zhangjiajie, China, Aug. 2019, pp. 1839–1844, doi: 10.1109/HPCC/SmartCity/DSS.2019.00253.

[20] *Corda Enterprise Version 4.3 Documentation*. Accessed: Mar. 19, 2020. [Online]. Available: https://docs.corda.r3.com/index.html

[21] M. J. M. Chowdhury, M. S. Ferdous, K. Biswas, N. Chowdhury, A. S. M. Kayes, M. Alazab, and P. Watters, "A comparative analysis of distributed ledger technology platforms," *IEEE Access*, vol. 7, pp. 167930–167943, 2019, doi: 10.1109/ACCESS.2019.2953729.

[22] J. S. Cuadrado, E. Guerra, and J. de Lara, "Static analysis of model transformations," *IEEE Trans. Softw. Eng.*, vol. 43, no. 9, pp. 868–897, Sep. 2017, doi: 10.1109/TSE.2016.2635137.

[23] L. Burgueno, J. Troya, M. Wimmer, and A. Vallecillo, "Static fault localization in model transformations," *IEEE Trans. Softw. Eng.*, vol. 41, no. 5, pp. 490–506, May 2015, doi: 10.1109/TSE.2014.2375201.

[24] M. Autili, A. Bertolino, G. De Angelis, D. D. Ruscio, and A. D. Sandro, "A tool-supported methodology for validation and refinement of early-stage domain models," *IEEE Trans. Softw. Eng.*, vol. 42, no. 1, pp. 2–25, Jan. 2016, doi: 10.1109/TSE.2015.2449319.

[25] N. Macedo, T. Jorge, and A. Cunha, "A feature-based classification of model repair approaches," *IEEE Trans. Softw. Eng.*, vol. 43, no. 7, pp. 615–640, Jul. 2017, doi: 10.1109/TSE.2016.2620145.

[26] H. M. Chavez, W. Shen, R. B. France, B. A. Mechling, and G. Li, "An approach to checking consistency between UML class model and its java implementation," *IEEE Trans. Softw. Eng.*, vol. 42, no. 4, pp. 322–344, Apr. 2016, doi: 10.1109/TSE.2015.2488645.

[27] M. Fleck, J. Troya, M. Kessentini, M. Wimmer, and B. Alkhazi, "Model transformation modularization as a many-objective optimization problem," *IEEE Trans. Softw. Eng.*, vol. 43, no. 11, pp. 1009–1032, Nov. 2017, doi: 10.1109/TSE.2017.2654255.

[28] H. Lu, S. Wang, T. Yue, S. Ali, and J. F. Nygard, "Automated refactoring of OCL constraints with search," *IEEE Trans. Softw. Eng.*, vol. 45, no. 2, pp. 148–170, Feb. 2019, doi: 10.1109/TSE.2017.2774829.

[29] R. Clariso, C. A. Gonzalez, and J. Cabot, "Smart bound selection for the verification of UML/OCL class diagrams," *IEEE Trans. Softw. Eng.*, vol. 45, no. 4, pp. 412–426, Apr. 2019, doi: 10.1109/TSE.2017.2777830.

[30] M. Núñez, D. Bonhaure, M. González, and L. Cernuzzi, "A model-driven approach for the development of native mobile applications focusing on the data layer," *J. Syst. Softw.*, vol. 161, Mar. 2020, Art. no. 110489, doi: 10.1016/j.jss.2019.110489.

[31] H. Moradi, B. Zamani, and K. Zamanifar, "CaaSSET: A framework for model-driven development of context as a service," *Future Gener. Comput. Syst.*, vol. 105, pp. 61–95, Apr. 2020, doi: 10.1016/j.future.2019.11.028.

[32] A. P. F. Magalhaes, A. M. S. Andrade, and R. S. P. Maciel, "Model driven transformation development (MDTD): An approach for developing model to model transformation," *Inf. Softw. Technol.*, vol. 114, pp. 55–76, Oct. 2019, doi: 10.1016/j.infsof.2019.06.004.

[33] S. Jacome and J. De Lara, "Controlling meta-model extensibility in model-driven engineering," *IEEE Access*, vol. 6, pp. 19923–19939, 2018, doi: 10.1109/ACCESS.2018.2821111.

[34] R. Hebig, D. E. Khelladi, and R. Bendraou, "Approaches to co-evolution of metamodels and models: A survey," *IEEE Trans. Softw. Eng.*, vol. 43, no. 5, pp. 396–414, May 2017, doi: 10.1109/TSE.2016.2610424.

[35] X. Xu, I. Weber, and M. Staples, "Model-driven engineering for blockchain applications," in *Architecture for Blockchain Applications*. Cham, Switzerland: Springer, 2019, pp. 149–174, doi: 10.1007/978-3-030-03035-3_8.

[36] T. Górski and J. Bednarski, "Modeling of smart contracts in blockchain solution for renewable energy grid," in *Computer Aided Systems Theory—EUROCAST* (Lecture Notes in Computer Science), vol. 12013, R. Moreno-Díaz, F. Pichler, and A. Quesada-Arencibia, Eds. Cham, Switzerland: Springer, 2020, pp. 507–514, doi: 10.1007/978-3-030-45093-9_61.

[37] Z. Gao, L. Jiang, X. Xia, D. Lo, and J. Grundy, "Checking smart contracts with structural code embedding," *IEEE Trans. Softw. Eng.*, early access, Feb. 3, 2020, doi: 10.1109/TSE.2020.2971482.

[38] N. Rozanski and E. Woods, *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*, 2nd ed. New Delhi, India: Pearson, 2015.

[39] T. Górski, "Architectural view model for an integration platform," *J. Theor. Appl. Comput. Sci.*, vol. 6, no. 1, pp. 25–34, 2012.

[40] T. Górski, "Verification of architectural views model 1+5 applicability," in *Computer Aided Systems Theory—EUROCAST* (Lecture Notes in Computer Science), vol. 12013, R. Moreno-Díaz, F. Pichler, and A. Quesada-Arencibia, Eds. Cham, Switzerland: Springer, 2020, pp. 499–506, doi: 10.1007/978-3-030-45093-9_60.

[41] *GitHub Repository With the Project of the UML Profile for Distributed Ledger Deployment*. Accessed: Mar. 19, 2020. [Online]. Available: https://github.com/drGorski/UMLProfileForDLT

[42] *Gradle Build Language Reference, Version 6.2*. Accessed: Mar. 19, 2020. [Online]. Available: https://docs.gradle.org/current/dsl/index.html

[43] *Human-Optimized Config Object Notation*. Accessed: Mar. 19, 2020. [Online]. Available: https://github.com/lightbend/config/blob/master/HOCON.md

[44] JSON. *Java Script Object Notation Home Page*. Accessed: Mar. 19, 2020. [Online]. Available: www.json.org/json-en.html

[45] J. Boyarsky and S. Selikoff, *OCP Oracle Certified Professional Java SE 11 Programmer I Study Guide: Exam 1Z0-815*. Toronto, ON, Canada: Wiley, 2020.

[46] *Project Lombok, Version 6.2*. Accessed: Mar. 19, 2020. [Online]. Available: https://projectlombok.org

[47] *GitHub Repository With the Project of the UML2Deployment Transformation*. Accessed: Mar. 19, 2020. [Online]. Available: https://github.com/drGorski/UML2Deployment

[48] *GitHub Repository of the UML2Deployment Plugin Transformation*. Accessed: Mar. 19, 2020. [Online]. Available: https://github.com/drGorski/UML2DeploymentPlugin

[49] Y. Cai, L. Xiao, R. Kazman, R. Mo, and Q. Feng, "Design rule spaces: A new model for representing and analyzing software architecture," *IEEE Trans. Softw. Eng.*, vol. 45, no. 7, pp. 657–682, Jul. 2019, doi: 10.1109/TSE.2018.2797899.

[50] *Systems and Software Engineering—Architecture Description*, International Standard ISO/IEC/IEEE 42010:2011, 2011. Accessed: Apr. 18, 2020. [Online]. Available: https://www.iso.org/standard/50508.html

[51] K. Gallaba and S. McIntosh, "Use and misuse of continuous integration features: An empirical study of projects that (Mis)Use travis CI," *IEEE Trans. Softw. Eng.*, vol. 46, no. 1, pp. 33–50, Jan. 2020, doi: 10.1109/TSE.2018.2838131.

[52] *GitHub Repository With the Transformation Validation Project*. Accessed: May 16, 2020. [Online]. Available: https://github.com/drGorski/UML2DeploymentCheck

[53] *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Standard 610.12-1990, 1990. Accessed: May 16, 2020. [Online]. Available: https://ieeexplore.ieee.org/document/159342?arnumber=159342

TOMASZ GÓRSKI (Member, IEEE) received the M.Sc. and Ph.D. degrees in computer science from the Military University of Technology (MUT), Warsaw, Poland, in 1997 and 2000, respectively. For ten years, he has served with the Computer Science Center, the General Staff of the Polish Armed Forces, and ended his military service as a major. After getting the Ph.D. degree, he has worked in many commercial software development projects, such as check-in system for the Polish Border Guard, and financial systems integration for Zone Vision Ltd., London. Since 2005, he has run his consulting firm RightSolution, IBM Authorized Training Provider for Rational brand. He has been an IBM Rational Certified Instructor for Rational Unified Process, Requirements Management, Object-Oriented Analysis and Design, and Java programming. From 2004 to 2017, he has worked as a Professor Assistant with MUT and the Build Center for Advanced Studies in Systems Engineering. He works as a Professor Assistant and the Head of the IT Systems Department, Polish Naval Academy, Gdynia. He teaches object-oriented programming. His research interests include software engineering, software architecture, model-driven engineering, and blockchain. He is the author of the Architectural views model 1+5, UML Profile for Integration Flows, Use Case API Design Pattern, and UML Diagram for Integration Flows. He shares his experience as the Program Committee Member of the International Conference on Systems Engineering and the Editorial Board Member of the *Open Computer Science Journal*. He is a member of the IEEE Computer and IEEE Systems, Man, and Cybernetics societies.

JAKUB BEDNARSKI received the M.Sc. degree in computer science from the Military University of Technology, Warsaw, Poland, in 2013. After graduation, he started working in many commercial software development projects related to the insurance market, such as business transformation program for the biggest Polish insurance company (PZU), a new claim process, and systems implementation for Polish and worldwide companies (Zurich Insurance, AXA). From 2015 to 2017, he has worked as a Lecturer with the Military University of Technology. Since 2017, he takes the position as the Senior Technical Lead in EY Company, where he is mainly responsible for delivery of the systems for the insurance market based on the Guidewire platform. Next to the IT consultancy position, since 2018, he takes the role of Research and Teaching Assistant with the Polish Naval Academy, Gdynia. He is an Enthusiast of the agile approach for software delivery. His research interests include software engineering, IT architecture, model-driven engineering, and blockchain.