Model-Driven Development of Distributed Ledger Applications

Piero Fraternali, Sergio Luis Herrera Gonzalez, Matteo Frigerio, and Mattia Righetti

Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Piazza Leonardo da Vinci 32, Milan, 20133, Italy. sergioluis.herrera@polimi.it

Abstract. The Distributed Ledger Technology (DLT) is one of the most durable results of virtual currencies, which goes beyond the financial sector and impacts business applications in general. Developers can empower their solutions with DLT capabilities to attain such benefits as decentralization, transparency, non-repudiability of actions and security and immutability of data assets, to the price of integrating a distributed ledger framework into their software architecture. Model-Driven Development (MDD) is the discipline that advocates the use of abstract models and of code generation to reduce the application development and integration effort by delegating repetitive coding to an automated model-to-code transformation engine. In this paper, we explore the suitability of MDD to support the development of hybrid applications that integrate centralized database and distributed ledger architectures and describe a prototypical tool capable of generating the implementation artefacts starting from a high level model of the application and of its architecture.

Keywords: Blockchain · Distributed Ledger · MDD · IFML

1 Introduction

Distributed Ledger Technology (DLT), popularized by the advent of virtual currencies, is having great impact also on applications outside the financial sector, such as those for the insurance industry, for the public administration, for NFT trading and more. The common trait of the business sectors in which DLT holds the greatest potential is the need of sharing data and transactions within a decentralized distributed network in a transparent yet secure way. Most of the time DLT functionalities must be integrated within a traditional application architecture. For example, an insurance business may start introducing the DLT technology in the claim payment workflow, while retaining a more traditional database-driven centralized architecture in the other processes. As the employment of DLT in businesses matures one can expect that companies will implement a migration strategy to progressively port, totally or in part, their business processes to this new architecture. This phenomenon resembles the transition to B2B application architectures in the nineties, when companies started migrating their business processes to the Web in the wake of the success

of B2C applications. Integrating centralized database-driven and decentralized ledger-driven architectures for developing hybrid applications poses new design and implementation challenges. At the conceptual level the boundary must be defined between the data and the operations that reside in either of the two platforms. At the physical level, suitable interfaces must be implemented between data and transactions in the distributed ledger and in the database. The development of hybrid DLT and database driven applications and business processes requires adequate methodologies and tools to reduce effort and cost, enforce uniform design patterns across projects, and ease the migration of workflows from one architecture to the other. One such methodology is Model-Driven Development (MDD), defined as the software engineering discipline that advocates the use of *models* and of *model transformations* as key ingredients of software development [14]. Abstraction is the most important aspect of MDD, which enables developers to create and validate a high level design of their application and introduce implementation-level architecture details at a later stage of the realization process. Implementation and architecture details are introduced via model transformations, which iteratively refine the high-level initial model, eventually getting to the final executable solution [2]. Nowadays MDD is applied in practice by the so-called low-code software development platforms[1], which exploit Platform Independent Modeling languages and code generators to automate the production of e.g., mobile, Web and enterprise applications. In this paper, we investigate the application of MDD to the development of hybrid applications that mix features of both centralized database-driven and decentralized ledger-driven architectures. The contributions of the paper can be summarized as follows:

- We introduce the class of hybrid DLT/DB applications, which consists of enterprise applications that implement data and processes on both distributed ledger and centralized database infrastructures.
- We describe the MDD process of hybrid DLT/DB applications in terms of inputs, activities and outputs. Specifically, we start from a development process scheme conceived for data intensive multi tier applications and discuss its extensions with activities that cope with the DLT requirements.
- We propose to model the design of hybrid DLT/DB solutions with a simple extension of the Domain Model and of the Interface and Action Models. For the sake of illustration, we express such models with OMG's UML Class Diagrams or Entity-Relationship Diagrams, OMG's Interaction Flow Modeling Language (IFML) diagrams, and an abstract action language for IFML [11]. The extension caters for the requirements posed by the integration of DLT into a traditional database-driven application development. The designer simply annotates the Domain Model entities and relationships to specify the platforms where such primitives are materialized. In this way, the operations of the Action Model that affect domain objects can infer the platform(s) in which the operations are executed. Similarly, the components of the Interface Model implicitly derive the content to publish in the interface from the appropriate data source.

[1] Examples of low code platforms are WebRatio, Mendix and Outsystems.

- We illustrate a prototypical version of a code generator mapping the high-level specifications of a hybrid DLT/DB application into the executable code for the Java EE architecture integrated with a popular DLT framework (Hyperledger Fabric). The code generator takes in input the Domain, Interface and Action Models and produces a fully functional multi-organization and multi-role hybrid DLT/DB application with Web/mobile GUIs.
- We showcase the MDD approach in the realization of the blueprint application of Hyperledger Fabric, a financial certificate trading solution.
- We discuss the integration of the developed modeller and code generator into the WebRatio[2] commercial low-code platform.

1.1 Running example

To illustrate the MDD of hybrid DLT/DB application, we exploit throughout the paper an exemplary application built on top of the blueprint Papernet network introduced the Hyperledger Fabric tutorial[3]. Papernet is a commercial paper network that allows participants to issue, trade and redeem commercial papers. The hybrid DLT/DB application will permit the authorized personnel of business companies to create commercial papers and share them in a DLT network, where they can be purchased and redeemed by the employees of financial trading companies. The hybrid nature of the application stems from the necessity to exploit a traditional centralized database architecture for storing the employee data, implementing role-based access control, recording in the company's own books both the details of the commercial certificates and other relevant accounting and internal auditing data (e.g., the certificate's creator). To such needs the DLT requirements add up: the operations on the papers must be implemented so as to ensure transparency, accountability and non-repudiability of operations.

2 Background

2.1 Distributed ledger technology and hybrid DLT/DB applications

Distributed ledger technology (DLT) is an approach for sharing data across a distributed network of participants with the guarantee of immutability of transactions. DLT evolved from the Peer-to-Peer (P2P), file sharing and blockchain technologies. In a P2P network the peers are connected computer systems and the assets are shared directly without a central server. In 2008 the Bitcoin virtual currency applied the P2P paradigm to financial assets [10]. The underlying *blockchain* data sharing technology opened the way to other P2P asset management frameworks, yielding to DLT as a general-purpose architecture. The blockchain is a specific type of DLT that uses cryptographic and algorithmic methods to create and verify a continuously growing append-only chain of

[2] https://www.webratio.com
[3] https://hyperledger-fabric.readthedocs.io/en/release-1.4/tutorial/commercial_paper.html

transaction blocks constituting the ledger. Additions are initiated by a node that creates a new block of data, e.g., containing several transaction records. Information about a new data block is shared across the network and all participants collectively determine the block validity according to a predefined algorithmic validation method. Only after validation all participants add the new block to their respective ledgers. With DLT no single entity in the network can amend past data and approve additions. An attacker willing to corrupt the ledger must gain control over the majority of the nodes.

The **Smart contracts** [15] are code packages deployed and executed in the nodes of a DLT network. They are programs that run, control or document relevant actions in the network. The source of a smart contract is stored in the blockchain, allowing any interested party to inspect its code and current state and verify its functionality; in this way, also the operational semantics of a smart contract cannot be changed without the consensus of the network participants. Smart contracts are replicated on all the network nodes; when a smart contract executes on a node, others can verify the result and the operations performed by the smart contract are recorded in the blockchain permanently.

A **Hybrid DLT/DB application** is a distributed application that: 1) has a client-server multi-tier architecture; 2) manages persistent data; 3) involves transactions that update data both in traditional database storage and in distributed ledger storage; 4) requires DLT-enabled transparent sharing and non-repudiability of operations; 5) exposes its functionality to the end-users through one or more client-side (web or mobile) interfaces; 6) optionally exposes its functionality to other applications through APIs (e.g., as Restful services).

The **State of a hybrid DLT/DB application** consists of two elements: the ledger state i.e., an immutable log of transactions and the world state, i.e., a database with business objects managed by application transactions. In a hybrid DLT/DB application the world state can be further distinguished into internal and external. This leads to a tripartite notion of the state of a hybrid DLT/DB application, in which each level has its specific update semantics:

- The *external world state* is the state of the objects in databases external to the network. It is updated by external operations and transactions, whose effect is not automatically recorded and shared in the ledger.
- The *internal world state* is the state of the objects recorded in the network data store. It is updated by smart contracts, whose effect is automatically recorded and shared in the ledger.
- The *ledger (or blockchain) state* is the state of the distributed ledger containing the log of the smart contract operations. It cannot be updated explicitly but only by the system as an effect of the smart contract execution. In other terms, the ledger state is read-only for the application business logic and can only used by applications to visualize smart contract execution history.

Permissionless vs Permissioned DLT DLT systems can be *permissionless* or *permissioned*. In permissionless systems such as those underlying the virtual currencies, the participants can join or leave the network at will, without being authorized by any entity. There is no central owner, and identical copies of the

ledger are distributed to all network participants. In permissioned DLT, members are pre-selected by someone, in general, an owner or an administrator of the ledger, who controls network access and sets the participation rules. Permissioned DLT systems have been conceived to support the use of the technology in business contexts in which the sharing of data and operations are constrained to a set of entities that satisfy the network access rules.

DLT Frameworks for application development The adaptation of DLT to general-purpose applications has led to the advent of software frameworks supporting the integration of DLT functions in business applications. *Hyperledger Fabric*[4] is an open-source enterprise-grade permissioned DLT platform with features such as participant identifiability, high transaction throughput, low transaction confirmation latency, and transaction privacy and confidentiality control. The architecture separates the transaction processing workflow into three different stages. The smart contracts, also called chaincode, comprise the system distributed logic for processing and agreement. The transaction ordering module and the transaction validation and commitment module implement the serialization and persistence of operations. Such separation reduces the number of verification levels, mitigates the network bottlenecks, improves network scalability and the overall performances. When a participant submits a transaction proposal, the network peers need to endorse it. When a majority of peers have agreed, an ordering service creates a block of transactions to be validated. After validation, the transactions are committed to the ledger. Since only confirming instructions, such as signatures and read/write sets, are sent across the network, the scalability and performance of the network are enhanced. The plug-in and component-based architecture of the platform also simplifies the reuse of existing features and the integration of custom modules.

2.2 MDD with the Interaction Flow Modeling Language

The Interaction Flow Modeling Language (IFML) [11] is an OMG standard for the platform-independent specification of interactive applications that allows developers to describe the organization of the interface, the content to be displayed, and the effect produced by the user interaction or by system events. The business logic of the actions activated by the user interaction can be modelled with any behavioral language, e.g., with UML sequence diagrams or with IFML extensions for action modelling [3].

Interface Structure. The core IFML element for describing the interface structure is the *ViewElement*, specialized into *ViewContainer* and *ViewComponent*. ViewContainers denote the modules that comprise the interface content. They can include ViewComponents which represent the actual content elements. Figure 3 shows the IFML specification of one of the interfaces of the Papernet application. The front-end comprises two ViewContainers: *Home* and *Issue Paper*, which in turn comprise the ViewComponents that specify their content. Different types of ViewComponents can be used to describe alternative content

[4] https://www.hyperledger.org/use/fabric

patterns. The basic ones are *Detail* ViewComponents that denote the publication of a single object, *List* ViewComponents that denote the publication of multiple objects, and *Form* ViewComponents that denote an input form. Depending on their type, ViewComponents can output parameters: a form has output parameters corresponding to the submitted values and a List ViewComponent has an output parameter that identifies the selected item(s). IFML shows the abstract source from which ViewComponents derive their content with a *DataBinding* element, which references an object class of the application Domain Model. The object(s) bound to a ViewComponent can be constrained by a *selector condition*, which can be parametric. For example, in Figure 3 the *Paper Details* ViewComponent displays the data of a *Paper* object. The selector condition `[creator = ?]` on the ViewComponent expresses a required input parameter corresponding to the identifier of the object to display. The parameter input-output dependencies between components are specified with *Flows*.

Events, Navigation Flows and Data Flows. ViewContainers and ViewComponents can be associated with *Events*, denoted as circles, to express that they support the user interaction. For example, a List ViewComponent can be associated with an Event for selecting one or more items, and a Form ViewComponent with an Event for input submission. The effect of an Event is represented by a *NavigationFlow*, denoted by an arrow, which connects the Event to the ViewElement/Action affected/triggered by it. IFML specifies (also implicitly) the input, output, and parameter passing from the source to the target of the NavigationFlow. For example, in Figure 3 the NavigationFlow from the *Issued papers* List ViewComponent to the *PaperDetails* Detail ViewComponent denotes that the user can interact with the list by selecting one item. Such an event determines the (re)computation of the content of the *Paper Detail* ViewComponent based on the identifier of the *Paper* object selected from the list. Input-output dependencies between ViewComponents can also be specified independently of interaction events, using *DataFlows* denoted as dashed arrows.

Actions. The above mentioned list selection Event expresses a user interaction which has the sole effect of updating the interface content. Events can also specify the triggering of business logic, executed prior to updating the state of the user interface. The IFML *Action* construct is represented by a hexagon symbol (see Figure 3) and denotes an invoked program treated as a black box, possibly exposing input and output parameters. The effect of an Event firing an Action and the possible parameter passing rules are represented by a NavigationFlow connecting the Event to the Action and possibly by DataFlows incoming to the Action from other IFML elements. The execution of the Action may trigger another Action, cause a change in the state of the interface and produce output parameters; this is denoted by termination events associated with the Action and connected by a NavigationFlow to the Action executed after it or to the ViewElement affected by it. In Figure 3 the outgoing NavigationFlow of the *Create Paper* Action is connected to the *Paper Details* ViewComponent and is associated by default with the output parameter of the Action (this is the identifier of the Paper object created by the Action, as shown Figure 4). When

the Action completes, the *Home* ViewContainer is displayed and the output parameter of the *Create Paper* Action determines the object shown in the *Paper Details* ViewComponent, i.e., the newly created Paper object.

3 Development of Hybrid DLT/DB applications

We adopt an enterprise application integration perspective whereby the specific DLT requirements add up to the requirements typical of enterprise applications. For this reason, we start from a development process scheme typical of multi-tier data-intensive applications and extend such a workflow with the inputs, activities and outputs needed to address the integration of DLT requirements.

The development process of data-driven enterprise applications encompasses the major phases of requirements specifications, design, implementation and maintenance/evolution. In this paper, we do not address the modalities in which such major concerns are addressed in a practical software life cycle (e.g., in a SCRUM agile development method) but rather focus on the input, output and tasks that characterize each concern to show how DLT requirements impact.

3.1 Requirement specification

Requirements specification collects and formalizes the essential knowledge about the application domain and expected functions. The input is the set of business requirements that motivate the application development and all the available information on the technical, organizational, and managerial context. The output is a document specifying what the application must do. In a traditional database-driven application the specifications typically comprises the identification of the user roles, the use cases of each role with pre-conditions, workflow and post-conditions, and a dictionary of the essential data. When DLT requirements come into play, the requirements specifications should also address the data and transaction sharing requirements. This can be done by identifying the organizations participating to the network to which the user roles belong and making explicit the operations of the use case workflows whose execution should be tracked in the network.

Running example Two types of organizations can join the Papernet network: *Issuers* are companies that create commercial papers to fund their operations, and *Traders* are financial organizations that transact such certificates as a form of investment. The application role models comprise *Issuer employees* and *Trader employees* who interact on behalf of their respective organizations. The relevant use cases comprise the *Issue paper* use case by the issuer employees and the *Buy Paper* and *Redeem Paper* use cases by the trader employees. The data dictionary comprises as entities the organizations, the organizations' roles, the users, the commercial papers and the issuing, buying and redeeming operations. The application publishes three user interfaces: one public interface for logging in, one protected interface for issuer employees and one protected interface for trader employees. Figure 1 shows a simplified and partial excerpt of the requirements specifications of the Papernet application.

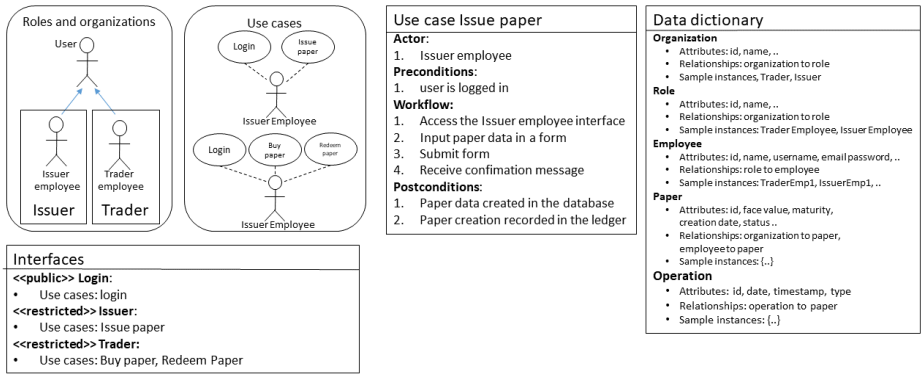

Fig. 1. The simplified requirements specifications of the Papernet application

3.2 Data design

Data design is the activity that takes in input the data dictionary and the use cases and creates the domain model of the application, using such notations as UML class diagrams or Entity-Relationship diagrams. In the design of the domain model of a hybrid DLT/DB application entities and relationships can be stereotyped with the three storage modes (`external`, `internal`, `ledger`) to show the level at which the information resides in the application state. The entities and relationships defined at the ledger level are implicitly read-only. The default storage mode is defined to be `external`, because it is assumed that only a minority of critical data will be stored in the internal database.

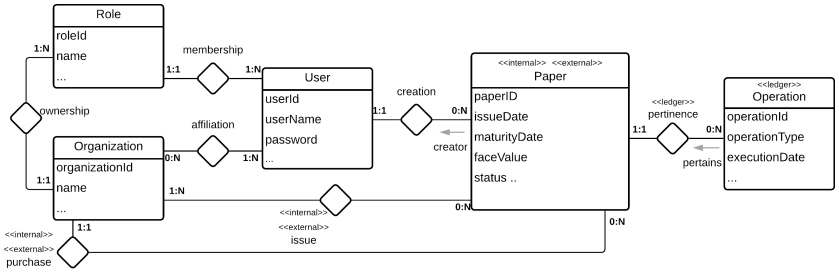

Fig. 2. Domain model of the PaperNet application

Running example The domain model of the Papernet application shown in Figure 2 contains five entities (Organization, User, Role, Paper, and Operation) and seven relationships (ownership, membership, affiliation, creation, issue, purchase and pertinence). The diagram shows which features are stored in the external database only (User, Role, Organization, membership, ownership, affiliation), which ones both in the internal and in the external database (Paper, issue

and purchase) and which ones in the ledger only (Operation and pertinence). Note that the information necessary for implementing role based access control is kept private in the external database of the organizations because authorization is needed also by other non DLT-enabled applications to install the enterprise access rights. Conversely, the information regarding the papers (including the reference to the creator and to the purchasing company) is preserved both in the internal and in the external database. They pertain to the external database because they serve administrative purposes and are also part of the internal state of the network because they enable the smart contract operations. Finally, the trace of the operations is maintained in the ledger only. This information is produced automatically by the DLT infrastructure and can be accessed in read-only mode by the application for inspecting the application history.

3.3 Interface design

Interface design maps the use cases into the IFML model of the GUIs that support the users' workflows. Each interface can be public or restricted to one or more roles and contains the ViewComponents, Flows and abstract Actions needed to support the associated use case(s). To disambiguate the case in which an entity or relationship belongs to multiple state levels, a default rule is introduced for the data binding of a ViewComponent. If no stereotype is used `external` is assumed. Otherwise `internal` or `ledger` can be specified.

Running example The Papernet application comprises one public and two restricted interfaces, as shown in Figure 1. The Issuer employee can access the interfaces shown in Figure 3. In the public interface a Form ViewComponent and a Login Action let the employee access the restricted interface. The restricted interface comprises a *Home* ViewContainer with a list of the commercial papers created by the employee. In the *Issued Papers* ViewComponent the data binding to the multi-level *Paper* entity refers by default to the external database and the selector condition `[creator = ?]` denotes a relationship join predicate evaluated in the external database. The DataFlow outgoing the predefined *GetUser* session ViewComponent is associated with the id of the logged in user as a parameter. Therefore the papers to show will be fetched from the external database by a parametric query that joins the Paper and the User entity on the *creation* relationship using the id of the logged in employee. By selecting one paper from the list, its data are shown in the *Paper Details* ViewComponent and the operations executed on it are displayed in the *Operations* ViewComponent. Note that in a real application, the *Paper Details* ViewComponent may show administrative attributes of the commercial paper that are pertinent to the business but not maintained in the network, which motivates the decision of binding the ViewComponent to the external database. A DataFlow between the *Paper Details* and the *Operations* ViewComponents expresses the input-output dependency between the two elements. In the *Operations* ViewComponent the data binding to the ledger *Operation* entity refers to this data layer and thus the selector condition `[pertains = ?]` also refers to a relationship predicate over the ledger. Therefore the operations to show will be fetched by retrieving

from the ledger the records related to the displayed paper. The interface also contains an *Issue Paper* ViewContainer, with a Form ViewComponent for inputting the data of a new paper. The Form comprises the fields corresponding to the attributes of the Paper entity (not shown in the IFML graphic notation). The submission of the *Issue* form (denoted by the event and NavigationFlow associated with this ViewComponent) triggers the execution of the *Create Paper* Action, whose internal workflow is specified in the action model of Figure 4.

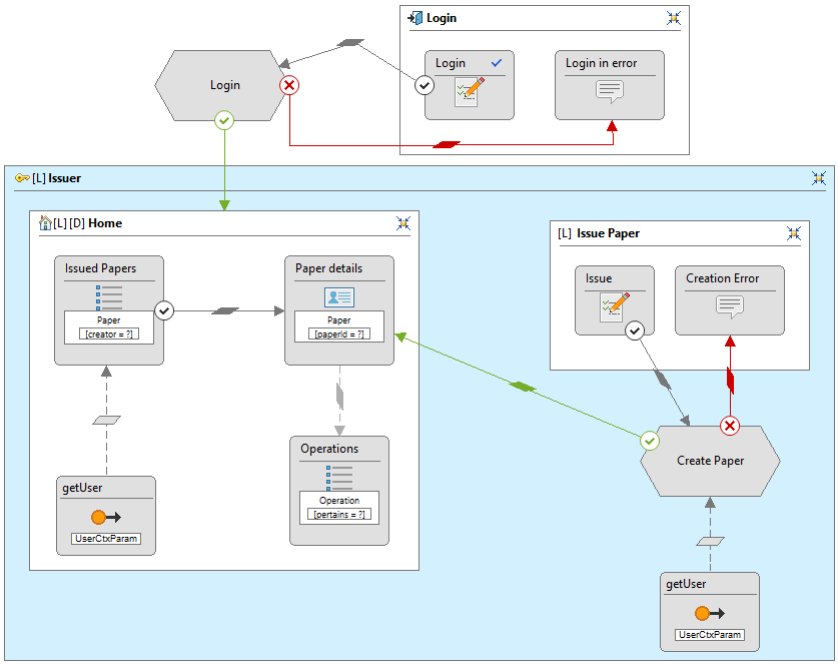

Fig. 3. IFML model of the public Login interface and of the restricted Issuer interface. The specification of the Create Paper abstract action is provided in Figure 4.

3.4 Operation design

Operation design maps each abstract Action into a detailed workflow. We express the workflow with the IFML action language extension of the WebRatio low code platform[5]. The action language comprises predefined operations, such as CRUD primitives bound to the entities and relationships of the domain model, system and session operations and more. When a CRUD operation affects a multi-level entity or relationship, the `internal` and `external` stereotypes can be associated with its data binding to disambiguate the data source. We assume `external` as

[5] https://www.webratio.com

the default. Given that applications have read-only access to the ledger, CRUD operations bound to ledger entities and relationships are not meaningful.

Running example The Papernet application comprises four operations (login, issue, buy, redeem) which are modelled as abstract Actions in the interfaces of the respective actor. Figure 4 shows the operation model of the *Create paper* abstract Action specified in the Issuer's interface (Figure 3). The workflow has an input port that specifies the parameters consumed by the Action. They are the creator's id, the face value and the maturity date (such parameters match the inputs to the abstract Action of Figure 3). The Action workflow starts by verifying the creator's credentials (with the *Verify User Identity* predefined read operation on the external database entity *User*); if this succeeds the current timestamp is acquired (with the predefined system operation *Get Time*); the timestamp and the input parameters are forwarded to the *Create Paper* operation, which is stereotyped as `internal` to denote that it is a smart contract creating a new paper instance in the internal database. If the update of the internal database succeeds, the issue operation is also automatically registered to the ledger. After the successful insertion of the paper, the workflow proceeds by updating the external database. The *Create Paper* operation stereotyped with `external` receives in input the attributes of the new commercial paper and stores them in the external database. The successful termination event of the `external` *Create Paper* operation is associated with a parameter (the identifier of the created object) which is bound to the success output port of the workflow. In the workflow of Figure 4 if any operation fails, the workflow terminates, and the Error output port is associated with a parameter that describes the failure. Workflows similar to that in Figure 4 are also specified for the actions of the Trader's role: paper purchase and redemption. We omit them for space reasons.

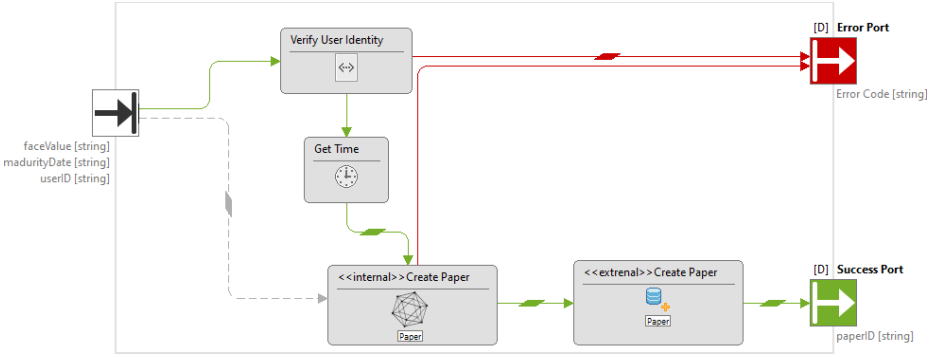

Fig. 4. Operation model of the Create paper abstract action

3.5 Architecture design

Architecture design describes at a high level the runtime infrastructure onto which the application is deployed. We assume a fixed architecture pattern consisting of a Java Enterprise back-end connected to one or more relational databases and a permissioned DLT network implemented with Hyperledger Fabric.

Running example Under the above mentioned assumptions, architecture design boils down to specifying the configuration parameters of the architecture of the specific application. These amount to the URLs and permissions to access the database by each organization and the network channels, peers and policies for each organization and for the channel. In the running example, we install a single *Trading* channel with two peers for each organization. Figure 5 represents the network model designed with the MDD tool described in Section 4.

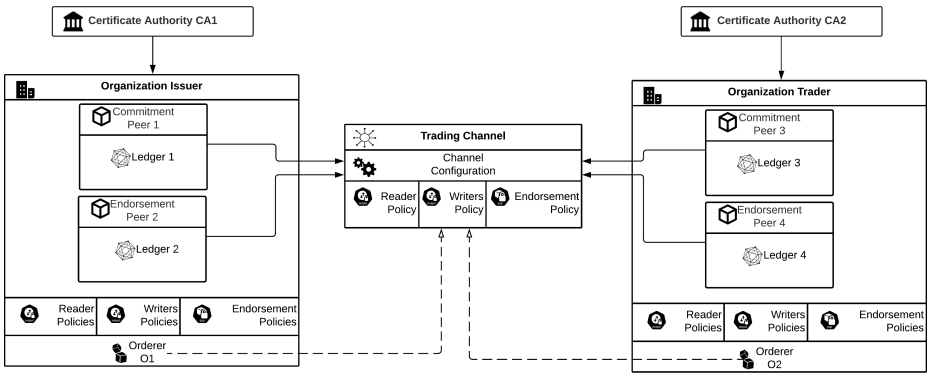

Fig. 5. Network Model

4 Implementation

To implement the described MDD approach, we extended the code generator of the WebRatio low code platform, which supports Entity-Relationship data modelling and IFML user interface modelling. The WebRatio code generator is template-based and enables developers to refine the implementation of the predefined components and to add new components and templates. We customized the behaviour of the data-driven ViewComponents to support the binding to internal database entities and relationships and added the action templates for CRUD operations bound to the internal database so as to map them to smart contracts. Furthermore, a completely new model editor and a deployment code generator were developed to map the network model into the artefacts required to install the network. Figure 6 illustrates the architecture, inputs and outputs of the MDD code generator of hybrid DLT/DB applications.

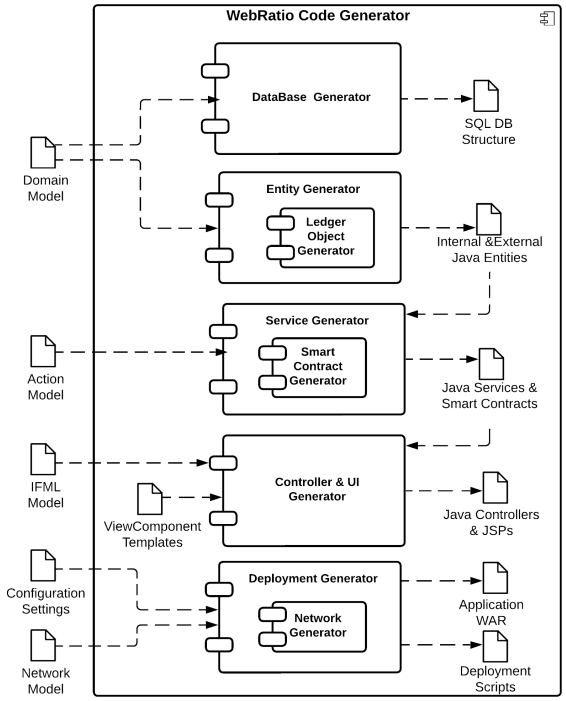

Fig. 6. Components of the code generator, including inputs and outputs

The **DataBase Generator** maps the domain model into SQL scripts that create the database structure. The native WebRatio module was reused as-is for mapping the external entities. No code generation is necessary for the Hyperledger Fabric store because all internal entities map to standard collections, and ledger entities map to JSON objects extracted via API calls.
The **Entity Generator** maps the domain model into Hibenate classes. The module has been extended with a *Ledger Object Generator* to map the internal database entities onto Java classes extending the Hyperledger Fabric Framework and the ledger entities onto JSON objects extracted from the blockchain.
The **Service Generator** takes in input the action model and the domain entities and outputs service classes that implement the operations and the control logic of the action workflows. It exploits code templates and generates the service API and the implementation code for the predefined system and CRUD operations. We extended the module by adding a *Smart Contract Generator*, which produces the smart contracts for managing the internal objects.
The **Controller and UI generator** takes in input the IFML models, the service interfaces and the predefined ViewComponent templates and outputs Java controllers and GUI views. We extended the module by adding code templates for the ViewComponents bound to the internal and ledger entities.
The **Deployment generator** takes in input the configurations defined in the

architecture model and creates a WAR package containing the generated code of the Web/mobile application. We extended the module by adding a *Network generator* that produces the artefacts for installing the network.

To model the network during architecture design, we defined a domain specific language with the Eclipse Modeling Framework (EMF)[6]. We identified the network entities, attributes, and relationships, represented them in an Ecore meta-model and built a graphical editor that supports the creation of network models. The Network generator, built using Acceleo[7], takes in input the network model and produces the following files for instantiating and starting the network automatically:

- configtx.yaml: contains the information to configure the channel(s);
- base.yaml: Docker-Compose file used to define a basic peer container configuration; it provides the standard actions for all the peer setup.
- dockercompose.yaml: Docker-Compose file used to define and run the Hyperledger Fabric containers that make up the network;
- dockercomposeca.yaml: Docker-Compose file used to define and run the Fabric-CA containers.
- create_channel.sh: script used to create a channel and to join the peers to it;
- deploy_chaincode.sh: script used to deploy a smart contract to a channel according to the Fabric chaincode life cycle.

5 Related Work

Efforts have been done to apply MDD to DTL applications. In [12] the author propose a method to model the behaviour of Etherium smart contracts using Entity-Relationship Diagrams, UML class diagrams, and BPMN process models. Marchesi et al. [8] discuss a design method for blockchain applications covering the definition of system goals, use cases, data structures, implementation of smart contracts and integration and testing. UML stereotypes are defined to account for DLT concepts. In [16,7] Business Process Models and data registry models are used as input for a code generator that creates smart contracts executable in the blockchain network. Corradini et al. [4] implement a similar approach for multi-party business processes interacting with multiple blockchain networks, by including a choreography model in which the target of each task is defined. Similar approaches for the generation of smart contracts can be found in [6] where enhanced state diagrams are used to represent the entities life-cycle as a set of states and transitions and a code generator takes the diagram as input and creates a smart contract for each transition. The FSolidM tool [9] employs a similar technique but uses finite state machines to represent states and transitions. In [5] the authors propose B-MERODE, an MDD methodology for blockchain applications that represents the application in 5 layers: Domain, Permission, Core Information System Services, Information System Services, and Business

[6] https://www.eclipse.org/modeling/emf/
[7] https://www.eclipse.org/acceleo/

Process. The models can be created using UML Class diagrams, finite state machines, BPM or tables. The method includes also compliance checks. However, no code generation is provided. The same authors propose the MDE4BBIS framework [13], an extension to B-MERODE that defines the transformations for generating smart contracts from activity models. CEPchain[1] is a tool to connect a blockchain to a Complex Event Process (CEP) Engine; the author use domain-specific languages to model the smart contract behaviour, the CEP domain, and the events that trigger processes. The models are translated into smart contracts and into event patterns code deployed on the network and on the CEP, allowing the CEP engine to trigger transactions on the blockchain when event pattern conditions are met. Most of the revised works focus on defining software engineering artefacts for modeling blockchain applications and helping developers in the design process. Some of them include code generation, but only focused on the generation of smart contracts, leaving the rest of the application development and network deployment to be performed manually. Our approach provides an MDD method for designing hybrid DLT/DB solutions and a platform that generates the complete code for the deployment of the network, the installation of the peers and of the smart contracts and a fully functional web/mobile application for interacting with the network and the ledger, internal and external data.

6 Conclusion

In this paper, we have described a novel MDD method and toolchain for supporting the development of so-called hybrid DLT/DB applications. The method has been implemented by extending a commercial MDD platform with features specific to DLT applications. The current implementation maps the models into a fully functional Hyperledger Fabric network, the smart contracts realizing the network operations and a web/mobile GUI for interacting with both database and ledger content. Future work will focus on improving the generated code for advanced non-functional requirements such as scalability and atomicity of transactions spanning the internal and external database and on testing the usability of the method and tools with developers of real-world applications.

Acknowledgement This work has been supported by the European Union's Horizon 2020 project PRECEPT, under grant agreement No. 958284.

References

1. Boubeta-Puig, J., Rosa-Bilbao, J., Mendling, J.: Cepchain: A graphical model-driven solution for integrating complex event processing and blockchain. Expert Systems with Applications **184**, 115578 (2021). https://doi.org/https://doi.org/10.1016/j.eswa.2021.115578, https://www.sciencedirect.com/science/article/pii/S0957417421009805
2. Brambilla, M., Cabot, J., Wimmer, M.: Model-Driven Software Engineering in Practice, Second Edition. Synthesis Lectures

on Software Engineering, Morgan & Claypool Publishers (2017). https://doi.org/10.2200/S00751ED2V01Y201701SWE004, https://doi.org/10.2200/S00751ED2V01Y201701SWE004
3. Brambilla, M., Fraternali, P.: Interaction flow modeling language: Model-driven UI engineering of web and mobile apps with IFML. Morgan Kaufmann (2014)
4. Corradini, F., Marcelletti, A., Morichetta, A., Polini, A., Re, B., Scala, E., Tiezzi, F.: Model-driven engineering for multi-party business processes on multiple blockchains. Blockchain: Research and Applications **2**(3), 100018 (2021). https://doi.org/https://doi.org/10.1016/j.bcra.2021.100018, https://www.sciencedirect.com/science/article/pii/S2096720921000130
5. De Sousa, V.A., Burnay, C., Snoeck, M.: B-merode: a model-driven engineering and artifact-centric approach to generate blockchain-based information systems. In: International Conference on Advanced Information Systems Engineering. pp. 117–133. Springer (2020)
6. Garamvölgyi, P., Kocsis, I., Gehl, B., Klenik, A.: 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). pp. 134–139 (2018). https://doi.org/10.1109/DSN-W.2018.00052
7. Lu, Q., Binh Tran, A., Weber, I., O'Connor, H., Rimba, P., Xu, X., Staples, M., Zhu, L., Jeffery, R.: Integrated model-driven engineering of blockchain applications for business processes and asset management. Software: Practice and Experience **51**(5), 1059–1079 (2021)
8. Marchesi, M., Marchesi, L., Tonelli, R.: An agile software engineering method to design blockchain applications. In: Proceedings of the 14th Central and Eastern European Software Engineering Conference Russia. pp. 1–8 (2018)
9. Mavridou, A., Laszka, A.: Designing secure ethereum smart contracts: A finite state machine based approach. CoRR **abs/1711.09327** (2017), http://arxiv.org/abs/1711.09327
10. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system. Decentralized Business Review p. 21260 (2008)
11. OMG: Interaction flow modeling language (IFML), version 1.0. http://www.omg.org/spec/IFML/1.0/ (2015)
12. Rocha, H., Ducasse, S.: Preliminary steps towards modeling blockchain oriented software. In: 2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB). pp. 52–57. IEEE (2018)
13. de Sousa, V.A., Burnay, C.: Mde4bbis: A framework to incorporate model-driven engineering in the development of blockchain-based information systems. In: 2021 Third International Conference on Blockchain Computing and Applications (BCCA). pp. 195–200. IEEE (2021)
14. Stahl, T., Völter, M.: Model-Driven Software Development: Technology, Engineering, Management. Wiley, Chichester, UK (2006)
15. Szabo, N.: Formalizing and securing relationships on public networks. First monday (1997)
16. Tran, A.B., Lu, Q., Weber, I.: Lorikeet: A model-driven engineering tool for blockchain-based business process execution and asset management. In: BPM (Dissertation/Demos/Industry). pp. 56–60 (2018)