Integrated Model-Driven Engineering of Blockchain Applications for Business Processes and Asset Management

Qinghua Lu\textsuperscript{a}, An Binh Tran\textsuperscript{a}, Ingo Weber\textsuperscript{b,1}, Hugo O’Connor\textsuperscript{a}, Paul Rimba\textsuperscript{a}, Xiwei Xu\textsuperscript{a}, Mark Staples\textsuperscript{a}, Liming Zhu\textsuperscript{a}, Ross Jeffery\textsuperscript{a}

\textsuperscript{a}Data61, CSIRO, Sydney, Australia
firstname.lastname@data61.csiro.au
\textsuperscript{b}Technische Universitaet Berlin, Germany
firstname.lastname@tu-berlin.de

Abstract

Blockchain has attracted a broad range of interests from start-ups, enterprises and governments to build next generation applications in a decentralised manner. A typical class of applications uses blockchain for the management of cross-organisational business processes as well as assets. However, developing such applications without introducing vulnerabilities or bugs is hard for developers, not the least because the deployed code is immutable and can be called by anyone with access to the network. Model-driven engineering (MDE) helps to reduce those risks, by combining proven code snippets as per the model specification, which is typically easier to understand than source code with all its implications. Therefore, in this paper, we present an approach for integrated MDE across business processes and asset management. Business processes and asset management are integrated in that business processes can control assets, and assets can make use of business processes, e.g., for settlement. Our approach includes methods for fungible/non-fungible asset registration, escrow for conditional payment, and asset swap. The input models comprise business process models and fungible/non-fungible registry data schemas, while the generated smart contracts consist of business process execution smart contracts and

\textsuperscript{1}Majority of the work done while this author was with Data61, CSIRO.
standardised ERC-20/ERC-721 compliant asset registry smart contracts. The proposed MDE approach is implemented in a smart contract generation tool called Lorikeet, and evaluated in terms of feasibility, functional correctness, and cost effectiveness.

**Keywords:** Blockchain, smart contract, model-driven engineering, business process, asset, registry

### 1. Introduction

Blockchain has attracted a wide range of interests from start-ups, enterprises and governments. Those interests have been sparked by the possibility of using blockchain as a general, decentralised and trustworthy compute environment through the advent of smart contracts. A large number of projects have been conducted to explore how to use blockchain to re-architect systems and to build new applications and business models [3].

A typical class of applications uses blockchain for the management of business processes across organisations as well as for digital asset management, which are maintained and controlled on-chain. Assets can be classified into fungible assets and non-fungible assets. Fungible assets are individual units that are interchangeable (e.g., company share and gold), while non-fungible assets represent unique assets (e.g., cars, patents, houses). Both fungible and non-fungible assets are traditionally managed by relying on a centralised trusted authority, which can cause trust issues and introduce inefficiencies or counterparty risks (e.g., re-assigning ownership of goods before payment). However, it is hard for developers to develop blockchain applications for business processes and asset management without introducing vulnerabilities or bugs, not the least because the deployed smart contract code is immutable and can be called by anyone with access to the network [13]. Model-driven engineering (MDE) [2] helps to reduce those risks, by combining proven code snippets as per the model specification, which is typically easier to understand than source code with all its implications.
Therefore, in this paper, as shown in Fig. 1, we present an approach for integrated MDE across business processes and asset management: business processes are supported, as are fungible (e.g., ERC20 tokens) and non-fungible (e.g., car/grain/land titles) assets. Business processes and asset management are integrated in that business processes can control assets, and assets can make use of business processes, e.g., for settlement. Previous efforts targeted MDE for business processes [26] and non-fungible assets [23] in isolation. In this paper, we expand the scope of the approaches, and tackle the non-trivial integration of business processes with both fungible and non-fungible asset management. The value of this integration is managing processes for both fungible and non-fungible asset registration, escrow for conditional payment, and asset swap in an efficient way using blockchain, which are not covered by our previous work. We design and develop a tool called Lorikeet\(^2\) that implements the proposed MDE approach. Our evaluation results show that the proposed approach is feasible and functionally correct.

The contributions of this paper are as follows.

- An MDE approach for development of blockchain applications for business

\(^2\)Rainbow Lorikeet is a species of parrots often encountered in Sydney, Australia – see https://en.wikipedia.org/wiki/Rainbow_lorikeet (accessed 26 May 2020)
processes and asset management.

- Modelling methods to specify models for integration of business processes with asset management, including both fungible and non-fungible asset registration, escrow for conditional payment, and asset swap. We provide the templates for the developers to customize data schemas for both fungible and non-fungible assets registries. We also extend the OMG standard Business Process Model and Notation (BPMN) 2.0 to specify interactions between business processes and fungible/non-fungible asset registries.

- Smart contract generation methods to automatically transform models into smart contract programming language. The input models include business process models, and fungible/non-fungible registry data schemas, while the generated smart contracts consists of business process execution smart contracts and standardised ERC-20/ERC-721 compliant asset registry smart contracts. Interactions between business processes and asset registries (e.g., for escrow or asset swap) are also implemented in the produced smart contracts.

- Blockchain interaction methods to handle compilation and deployment of smart contracts and communication with the deployed smart contracts on blockchain.

  • Feasibility and functional correctness evaluation using four industrial use cases which cover fungible and non-fungible asset registration, escrow for conditional payment, and asset swap.

  • An analysis of gas consumption and comparison with numbers from over 292 million transactions on the public Ethereum blockchain.

The remainder of this paper is organised as follows. Section 2 discusses the background and related work. Section 3 presents our MDE approach. Section 4 introduces our tool named Lorikeet. Section 5 evaluates the proposed approach using use cases. Section 6 concludes the paper and outlines the future work.
2. Background and Related Work

In this section, we first introduce blockchain technology and smart contracts in Section 2.1. Then, we provide background knowledge of Model-Driven Engineering (MDE) and its benefits in Section 2.2. Finally, we explain why MDE and blockchain can be a solution for addressing the trust issue in the business process domain (Section 2.3) and the asset registry domain (Section 2.4), which has not been fully solved before.

2.1. Blockchain and Smart Contracts

A blockchain is an append-only store of transactions distributed across computational nodes and structured as a linked list of blocks, each containing a set of transactions. The main purpose of structuring the data store into blocks is to obtain manageable chunks of information, for communication as well as for achieving consensus. Blockchain was introduced as the technology behind Bitcoin [15]. Its concepts have been generalized to distributed ledger systems that verify and store any transactions without coins or tokens [24], without relying on any central trusted authority like traditional banking or payment systems. Instead, all participants in the network can reach agreements on the states of transactional data to achieve trust.

Merkle trees are an important part of blockchain, supporting fundamental blockchain functionality and enabling efficient and secure verification of large data structures. Merkle trees have a hash-based structure that can ensure data integrity in a trivial way: each node (except leaves) in the tree contains the hash of its child node values; if nothing changed, the root will be the same; otherwise only the hashes on the path from the root to the changed leaves are changed.

A smart contract is a user-defined program that is deployed and executed on a blockchain system [16], which can express triggers, conditions and business logic [26] to enable complex programmable transactions. Smart contracts can be deployed and invoked through transactions, and are executed across the blockchain network by all connected nodes. The signature of the transaction

5
sender authorizes the data payload of a transaction to create or execute a smart contract. Trust in the correct execution of smart contracts extends directly from regular transactions, since (i) they are deployed as data in a transaction and thus immutable; (ii) all their inputs are through transactions and the current state; (iii) their code is deterministic; and (iv) the results of transactions are captured in the state and receipt trees, which are part of the consensus.

When using a blockchain, there are different types of deployments, including public blockchain, consortium blockchain or private blockchain. Public blockchains, which can be accessed by anyone on the Internet ("permissionless"), have high information transparency and auditability, but sacrifice performance and a cost/incentive model. A consortium blockchain is typically used across multiple organisations and the rights to read/write on the blockchain may be restricted to specific participants. In a private blockchain network, write permissions are often kept within one organisation, although this may include multiple divisions of a single organisation. Private blockchains are the most flexible for configuration because the network is governed and hosted by a single organisation. A blockchain may be permissioned in requiring that one or more authorities act as a gate for participation. This may include permission to join the network and read information from the blockchain, to initiate transactions, or to create blocks. Permissions can be stored either on-chain or off-chain. There are often tradeoffs between permissioned and permission-less blockchains including transaction processing rate, cost, censorship-resistance, reversibility, finality and flexibility in changing and optimising the network rules.

2.2. Model-Driven Engineering

Model-driven engineering (MDE) is a methodology that uses models at various levels of abstraction to address software development complexity [17]. Domain-specific MDE can help map the model of the problem domain to the design of the software solution [7? ]. The abstraction level for models can be at various degrees. For example, some models can directly derive the production code while others can only be used to guide the developers in developing the
software. In model-driven engineering, models can produce code or guide implementation, or conversely, the code (or other artefacts) can generate models to help understand the software design, e.g. database schema. Depending on the purpose, various concepts can be captured in models, e.g. system/database structures or a sequence of activities.

Specifically, MDE for code generation can be further classified into different types:

- **Once-off code generation**: Once the code is derived from the model, the subsequent evolution of the code is independent of the model.

- **Repetitive code generation**: The code is re-generated from the model following subsequent changes to the model over time.
  - **One-way model-to-code code generation**: The code is updated if changes are made to the model, but not vice versa.
  - **Round-trip code generation**: If the generated code is updated, the changes can be propagated back to the model level. This is an often desired but rarely achieved vision for MDE.

In the context of blockchain-based applications, MDE is of particular importance for the following reasons [12]. First, model-driven engineering tools can implement best practices and generate well-tested code, thereby avoiding vulnerable code which may be easily attacked (e.g. the DAO exploit on the Ethereum blockchain\(^3\)). Second, models can avoid lock-in to specific blockchain technologies since they can be platform-agnostic, and a model-driven engineering tool might be able to produce artefacts for multiple blockchain platforms. Third, models are easier to understand than code, thus improving the development productivity. It is easier to check the correctness of a model and MDE tools can ensure that the deployed code has not been modified after its generation from the model. Fourth, it can facilitate communication with domain

\(^3\)http://www.coindesk.com/understanding-dao-hack-journalists
experts since domain experts can look at the model to understand how their ideas are represented in the system.

2.3. Business Processes

Trust issues in business processes [25] have been discussed over the last decade. [4] uses selective encryption and restricts data access for both the broker and the service partners to achieve trust with untrusted broker. Mont and Tomasi [14] design a trust service for cross-company collaboration based on a hybrid architecture mixing a trusted centralized control with untrusted peer-to-peer components. [10] present an agent-based architecture that can remove the scalability bottleneck of a centralized orchestration engine and provides more efficiency by executing portions of processes close to the data they operate on. [19] select partners on the basis of disclosure policies and credentials (i.e. identity attributes issued by a “Credential Authority”) in virtual organisations. Various important concepts such as conformance [1], reliability [21] and quality of services [29] have been studied for centrally controlled business process execution.

However, these works do not solve the trust issue as a collaborating party might have corrupted their historic files to their advantage. There is no party that sees all the messages in the business processes. Technologies such as shared data stores provide solutions via consensus protocols to synchronize replicas [9] in a fully trusted environment. Therefore, our work focuses on using blockchain to address the trust issue in the business process domain and the asset registry domain. The tool Caterpillar [11] is a Business Process Management System (BPMS) operating on blockchain, and does not support asset management. In contrast, Lorikeet is an MDE tool that addresses the hard challenge of integrating asset registry with business processes on blockchain.

2.4. Registries

A registry is a list of data recorded and maintained by a trusted authority, which is an authoritative database for specific entities and is used to manage
many aspects of daily life, such as land titles, business names, books, marriages, births and deaths, music, films and domain names. Traditionally, registries are maintained by a central authority. However, such centralised architecture may cause a single point of failure for the whole registry system. Building registries on a blockchain [23] can guarantee data integrity, availability, transparency and immutability, which are key requirements for registries [6]. Additionally, blockchain can be used as a unified infrastructure which enables multiple registries to easily interact with each other.

There are registries being built on blockchain in ad-hoc ways, for example, Namecoin⁴, which is a domain name registry that shares the same network with Bitcoin⁵, and Abscribe⁶, which is an artwork registry that enables artists to maintain the ownership of their digital artwork. However, building a registry on blockchain is non-trivial since developers need to understand in depth how particular blockchain platforms operate and learn smart contract programming languages. Regis⁷ is a smart contract generation tool on Ethereum⁸ blockchain, but only provides basic operations. We introduced our registry generator tool for blockchain in [23] and briefly discussed how to integrate registries with business processes in a demo paper [22].

3. An Integrated Model-Driven Blockchain Application Development Approach for Business Processes and Asset Management

In this section, we present our Model-driven blockchain application development approach for business processes and asset management. We first provide an overview of our MDE approach in Section 3.1. Then, we discuss the modelling methods proposed for fungible/non-fungible asset registries and extensions of BPMN to support modelling of interactions between business processes and

⁴https://namecoin.org/
⁵https://bitcoin.org/
⁶https://www.ascribe.io/
⁷https://regis.nu/
⁸https://www.ethereum.org/
asset registries in Section 3.2. After that, we propose the methods for business process smart contract translation and registry smart contract generation in Section 3.3. Finally, we explain the blockchain interaction methods for connecting with a blockchain node, and handling the compilation, deployment as well as communication with smart contracts in Section 3.4.

3.1. Overview of the Model-Driven Engineering Approach

Fig. 2 illustrates an overview of our model-driven engineering (MDE) approach for integrating business processes with asset management on blockchain. The design of the approach consists of three parts: modelling, smart contract (SC) generation, and blockchain interaction. For modelling, the approach provides templates for the developers to customize fungible/non-fungible asset registry data schemas and extends BPMN 2.0 to support modelling of interactions between business processes and asset registries (e.g. for fungible/non-fungible asset registration, escrow for conditional payment, and asset swap). The approach then transforms the built models (i.e. models for business processes, asset registries, and their integration) into blockchain smart contract implementations in a programming language (such as Solidity) and handles interaction with smart contracts deployed on blockchain. There are three types of users for this MDE approach (and the corresponding tool): 1) developers can use it to improve development productivity and quality, 2) operators can use it to monitor the execution of generated smart contracts; 3) and domain experts can use it to communicate with developers and understand how their ideas are represented in the system.

3.2. Modelling

As shown in Fig. 2, we propose modelling methods for business processes, asset registries, and their interactions. Business process modelling includes process modelling in BPMN 2.0 and interaction modelling using the newly extended BPMN elements. For registry modelling, we provide different methods for modelling fungible assets (e.g. ERC20 tokens) and non-fungible assets (e.g. car/grain/land titles) respectively.
3.2.1. BPMN Modelling

In addition to standard BPMN modelling, model-level integration of business processes with asset management requires interface specification of smart contracts including asset/data registry smart contracts and escrow smart contracts. A smart contract on a blockchain, among others, acts as a data store which tasks in a business process can read data from, or write data to. However, the existing BPMN 2.0 elements (i.e. DataStoreReference and DataOutputAssociation) do not support representation of properties specific to smart contracts (e.g., registry smart contract addresses and smart contract invocation). Thus, to support integration, we introduce custom elements to BPMN 2.0 and design respective graphical representations for them, which are shown in Table 1.
Table 1: Custom BPMN Elements

| Element                          | Description          | Notation |
|---------------------------------|----------------------|----------|
| bcext                           | BPMN meta-model name space | N/A      |
| SmartContractInterface          | Smart contract element | ![Image](image.png) |
| ConnectionOutgoingContractInvocation | Connection to the smart contract | ..........> |

The elements include `bcext`, `SmartContractInterface`, `ConnectionOutgoingContractInvocation`. `bcext` is the custom name space for blockchain smart contract relevant BPMN meta-model. `SmartContractInterface` represents an interface for a smart contract which is external to the business process. `ConnectionOutgoingContractInvocation` is the custom connection which links business process tasks with the external smart contract in the BPMN model. The graphical notation for `SmartContractInterface` is extended from the existing BPMN notation for `DataStoreReference`, while the graphical notation for `ConnectionOutgoingContractInvocation` is designed extending the current BPMN notation `DataOutputAssociation`.

Fig. 3 shows the data structure of the proposed BPMN meta-model for smart contract-relevant extensions. `SmartContractInterface` can represent any type of smart contracts, e.g., asset/data registry smart contract, escrow smart contract, etc. When a smart contract is deployed on blockchain, it is uniquely identified and reachable via a smart contract address. In order for the business process smart contract to interact with the smart contract, the smart contract address can be made available to the BPMN model via the attribute `contractAddress` in `SmartContractInterface`. If the smart contract address is provided, the address is fixed for all the instances of this business process and cannot be changed. If it is not provided in the model, the translated business process smart contract allows users to specify the registry contract address each time a new business process instance is created. Each `SmartContractInterface` can have multiple `SmartContractFunctions`, which are provided to inform the
BPMN model of which functions are available to interact with a given smart contract and how to invoke each smart contract function. Each SmartContractFunction has one of each FunctionInputParameters and FunctionOutputParameters. Each of FunctionInputParameters and FunctionOutputParameters consists of many FunctionParameters. To enable BPMN tasks to communicate with the smart contract reference (i.e. retrieving data from and writing data to a smart contract), each SmartContractInterface can have many ConnectionOutgoingContractInvocations. As aforementioned, interaction with a smart contract is performed via invocations of smart contract functions. Therefore, ConnectionOutgoingContractInvocation specifies the signature of the smart contract function to be invoked by the BPMN task via the attribute fnName. Each ConnectionOutgoingContractInvocation has one of each FunctionInputBindings and
FunctionOutputBindings element. Each of FunctionInputBindings and FunctionOutputBindings contains multiple ParameterBindings. Using the attribute values in inputParameters and outputParameters, the model can specify bindings from the business process internal variables, or the BPMN task’s own input parameters (if it is a user task), to the input parameters and return values of the smart contract function, respectively.

3.2.2. Registry Modelling

On the registry side, we provide modelling methods for users to design fungible and non-fungible assets via the respective data registry template in UML. Fig. 4 shows the data model for fungible asset registry, which consists of basic token details and advanced token features. The basic token details include token name, symbol (an abbreviation, like “ETH” for Ether, usually 3 or 4 characters in length), and decimals (the number of digits in the fraction part).

The advanced features describe token design details about minting (isMintable, minterAddresses), burning (isBurnable, burnerAddresses), and initial distribution (initiallyDistributedAccounts). Users can configure the accounts that can mint or burn the token, the total supply of the token (the total number of tokens that have been or will be mined), and the accounts that receive the initial distribution and the amount sent to each account.

Fig. 5 illustrates the data structure for non-fungible asset registry, which specifies basic information and advanced features of non-fungible assets. Basic
information includes registry name, type, and user-defined attributes. Registry type can be ‘single’ or ‘distributed’. The ‘single’ registry type holds all records as values in the data store as a singleton registry smart contract, which is suitable for simple registries. The ‘distributed’ registry type manages each record as a separate smart contract, which is suitable for registries with complex operations, such as individual record-level permission management. A main registry smart contract creates these contracts and stores pointers to them. Regarding user-defined attributes, users can specify attribute name, type, whether a record is updatable, and maintain a detailed history of changes made to every registry record.

The non-fungible asset registry modeller also supports advanced features including record lifecycle management and access control. The registry record lifecycle (create, read, update, and delete) can be enabled to be managed via a business process executed on the blockchain. In this case, only the business process instance is allowed to create or update records, even though the registry is readable by the public (isOwnershipTransferEnabled, isRecordCreationRestrictedToBPMN, isOwnershipTransferEnabledToBPMN). For example, transferring ownership can be restricted to a business process instance, which exchanges ownership of the registered items, such as grain title. Access control can be enabled on registry functions (isRegistryFunctionAccessControlEnabled) and individual registry records (isRegistryRecordAccessControlEnabled), which can be implemented within the registry smart contract or using a separate smart contract (isAccessControlBySmartContractEnabled).

3.3. Smart Contract Generation

Smart contract (SC) generation consists of business process SC translation and registry SC generation. We previously presented our basic business process SC translation algorithm in [26, 27] and basic registry SC generation in [23]. Thus, in this section, we mainly discuss the extensions of SC generation, which includes extensions for ERC-20/ERC-721 standard compliance support, and integration between business process execution and registry.
| NonFungibleAssetRegistry |
|--------------------------|
| name : String            |
| attributes : Attribute[] |
| isRecordUpdatable : Boolean |
| isRevisionRecorded : Boolean |
| type : String            |
| isOwnershipTransferEnabled : Boolean |
| isRecordCreationRestrictedToBPMN : Boolean |
| isOwnershipTransferEnabledToBPMN : Boolean |
| isRegistryFunctionAccessControlEnabled : Boolean |
| isRegistryRecordAccessControlEnabled : Boolean |
| isAccessControlBySmartContractEnabled : Boolean |

| Attribute |
|-----------|
| type : String |
| name : String |

Figure 5: Non-fungible asset registry data model.

ERCs (Ethereum Request for Comments) are standards for developing Ethereum blockchain applications, e.g. token standards and name registries. To improve the interoperability between Ethereum blockchain applications that work with fungible assets and non-fungible assets, we extend the registry SC generation to produce ERC-20/ERC-721 compliant asset registry smart contracts which follows the rules and methods in the ERC-20\(^9\) and ERC-721\(^10\) interface. In other words, the functions in the smart contracts generated by fungible/non-fungible asset registry SC generation contain all the required ERC-20/ERC-721 functions respectively.

To support the interaction between business process execution smart contracts and registry smart contracts, SC generation is extended as illustrated in Algorithm 1. For each SmartContractInterface element in the business process model, the algorithm produces a smart contract interface in Solidity with all public functions specified in the element. If the contract address is set in the SmartContractInterface, the algorithm generates a variable containing that hard-codes the address for this smart contract interface. Otherwise, the algorithm outputs a process smart contract constructor parameter for setting the referenced smart contract address to a variable. For each ConnectionOutgoing-

---

\(^9\)https://eips.ethereum.org/EIPS/eip-20
\(^10\)https://eips.ethereum.org/EIPS/eip-721
Algorithm 1: Integration between business processes and registries

```plaintext
for each SmartContractInterface element do
  generate a Solidity smart contract interface with specified public functions
  if contractAddress is set then
    generate a variable containing hard coded address
  else
    generate a process contract constructor parameter for setting referenced smart
    contract address to a variable
  end
end
for each ConnectionOutgoingContractInvocation element do
  obtain source task, target smart contract interface
  generate Solidity code for invoking given target smart contract interface function
  while mapping inputs and return parameters to process variables as specified
  inject generated invocation code to function body of source task
end
```

The approach provides blockchain interaction methods for connecting with a blockchain node, and handling the compilation, deployment as well as communication with smart contracts. Thus, users can monitor the execution of smart contracts and interact with smart contracts directly. Specifically, the communication module comprises sending blockchain tractions, querying smart contract states, and listening to transaction progress and smart contract events. Sending blockchain transactions corresponds to write operations, while querying smart contract states applies to read operations. The communication module obtains
status of transactions and receives smart contract events via the listening to transaction progress and smart contract events.

Once the smart contracts are successfully deployed, the users can use the smart contracts to execute the business process instances and create records in fungible/non-fungible asset registries. The approach can furthermore generate user interface (UI) elements to interact with deployed smart contracts. Through these UI elements, users can execute smart contract functions as well as monitor smart contract events. Function invocation user interface forms are automatically populated from smart contract interfaces. Users can retrieve previously emitted events or continuously listen to new events. Process flow conformance is enforced by the approach: if an invocation to execute a task is in conformance with the current process execution state, it takes place; otherwise the invocation is unsuccessful. If a process task was successfully executed, a new event is emitted and the state is updated; if unsuccessful, the information of this invocation is stored but the process state does not change.

4. Lorikeet: An Integrated MDE Tool for Development of Blockchain Applications

We design and develop a model-driven engineering tool for business processes and asset management on blockchain, named Lorikeet. As illustrated in Fig. 6, the tool consists of the user interface (UI) components and back-end components that are designed adhering to a microservice architecture.

The UI components are presented as web applications for users to build business process and registry models, and interact with the smart contracts. The business process (BP) modeller is extended from the bpmn-js modelling library\(^\text{11}\). The user can model fungible/non-fungible asset registry references and action invocation easily by dragging and dropping in the modeller UI and providing the relevant registry information (e.g. registry smart contract ad-

\(^\text{11}\)https://github.com/bpmn-io/bpmn-js
dress, available registry actions, parameter binding information, etc.). The fungible/non-fungible asset registry modeller provides a form for users to fill in the information required by the registry model. The users can compile, deploy, and interact with the smart contracts in the blockchain interactor. The supported interactions include deployment and execution of smart contract functions as well as monitoring of smart contract events. The smart contract function invocation UI forms are automatically populated from the respective smart contract interface, while the smart contract event monitoring UI can display all previously emitted events or list any events by continuously listening to the blockchain. The blockchain interactor component is written in TypeScript with Node.js version 10, implementing the REST API using express.js server. Users can securely create and manage their identities via MetaMask\(^{12}\) which connects to the blockchain interactor.

Back-end components, including business process (BP) smart contract (SC) translator, ERC721 asset registry SC generator, and ECR20 token registry SC generator, are built and deployed independently as Docker containers. BP SC translator automatically generates smart contracts from BP models, while

\(^{12}\)https://metamask.io/
ERC721 asset/ERC20 token registry generator derives smart contracts based on the registry models. For Ethereum, the smart contracts are written in Solidity, compiled with Solidity compiler version 0.4.24. We used Truffle framework v12 to compile and test smart contracts.

The BP/registry modeller and blockchain interactor communicate with the back-end microservices via an API gateway. The API gateway forwards API calls from the UI, such as translating BPMN model to smart contract code, to the corresponding microservice. In addition, the blockchain interactor sends information about the emitted events in real-time using socket.io.

Fig. 7 shows the business process modeller UI of our tool. Once the user makes changes to the BP model on the left-hand side, it is translated to the corresponding smart contract code which is displayed on the right-hand side. The UIs for ERC721 asset/ERC20 token registry modelling are similar to the BP modeller UI: in both cases there is a form on the left-hand side collecting the customised registry information.
5. Evaluation

To evaluate the feasibility, functional correctness, and cost of our model-driven engineering approach, we use Lorikeet to model four use cases and generate smart contract code based on the models. The selected use cases include initial coin offering (ICO) process, quality tracing process, task outsourcing process, and grain title creation process. They cover fungible/non-fungible asset registration, escrow for conditional payment, and asset swap respectively. We encountered these use cases in projects or discussions with industry or government, and modelled them based on publicly available information. According to the classification of experiments by Zelkowitz and Wallace [28], this evaluation falls into the category of case studies since we do not have control over the experimental conditions: the processes are taken from external units (industry or government).

5.1. Feasibility

In this section, we evaluate the feasibility of our MDE approach using four uses, including ICO process (Section 5.1.1), quality tracing process (Section 5.1.2), task outsourcing process (Section 5.1.3), and grain title creation process (Section 5.1.4).
5.1.1. ICO Process

An Initial Coin Offerings (ICO) is a way to raise funds for new projects using cryptocurrency: essentially, investors can buy tokens for a new startup / initiative / project and pay for those using an established cryptocurrency like Ether or Bitcoin. ICOs can be considered as fungible asset registration use cases, where the new token is the fungible asset. Fig. 8 shows an ICO business process model built using Lorikeet. The process starts with setting up the token exchange rate with Ether (i.e. ETH) and caps (including one soft cap and one hard cap). Once the token exchange rate and caps are determined, investors deposit ETH to the business process smart contract. If the soft cap is reached, the bought amount of tokens are transferred to the investors’ accounts, which are recorded in the token registry. The registry smart contract address and actions to be invoked for the registry are specified as attributes of SmartContractInterface.

On the registry side, we built a fungible asset registry model for the token registry by filling the form provided by Lorikeet. The token name is Lorikeet Coin, while the symbol is LRK. The decimal value we put is 2.

Lorikeet creates a smart contract for instantiating the ICO process taking the business process model as input, while generating a token registry smart contract using the specified token information in the form as fields and the ERC-20 standard methods as methods. The token registry smart contract interface defines the methods interacting with the corresponding token registry smart contract while the process monitor smart contract implements the business process instance.

The results show that Lorikeet can automatically generate an ICO business process smart contract and a ERC-20 standard complaint ICO token registry smart contract using the ICO BPMN model and token registry data schema respectively. The output business process smart contract checks whether business process instances run compliant with the BPMN model (e.g. only when soft cap is reached, LRK tokens are distributed), while the token registry smart contract maintains the tokens in each investor’s account. Instead of writing those two
complex smart contracts, developers only need to focus on designing and refining high-level models, which are easier to explain to ICO stakeholders and to check for correctness. Also, as the token registry smart contract is ERC-20 complaint, it can be easily integrated with other blockchain applications. The results show that Lorikeet can successfully support MDE of business processes and registries concerning fungible assets.

5.1.2. Quality Tracing Process

Fig. 9 illustrates the quality tracing process models for import commodities in China [5] built using Lorikeet, which can be viewed as a non-fungible asset registration use case. The quality inspection agency provides quality tracing services and issues a traceability certificate of commodity if all requirements are fulfilled. The process starts when a product supplier lodges a quality tracing application for each batch of products to the quality inspection company. The administrator processes the paper work (e.g. invoices) and payment. Then the agency assigns a factory examiner to check the factory address, production capability, quality control process, etc. After inspecting the factory, a freight yard examiner is sent to check the products on freight yard and inspect on-site loading. The examiner attaches lead seals to the product containers if the on-site loading processes meet requirements. In the meantime, a product sample is sent to a lab for sample testing. Once the application passes the inspections and testing, the agency issues the supplier a traceability certificate of commodity. All the relevant traceability information and certificates are stored in the traceability registry.

The quality tracing process and traceability registry are modelled in a similar way as ICO use case discussed in Section 5.1.1. The registry smart contract address and actions to be invoked for the registry are specified as attributes SmartContractInterface.

On the registry side, we built a non-fungible asset registry model for the certificate registry by filling the Lorikeet registry template. The asset name is CertificateOfOrigin and the registry type is single. The record ID repre-
Figure 9: Quality tracing process model built using Lorikeet.

Lorikeet outputs a smart contract for creating a quality tracing process instance based on the built business process model and generates a ERC-721 compliant smart contract for the traceability certificate registry based on the built registry model.

The generated quality tracing process smart contract can check whether the process instances are executed consistently with the BPMN models (e.g. factory inspection, sample tests, and on-site loading supervision must be done before issuing a certificate), while the certificate registry smart contract maintains the hash values of certificates. Developers only need to specify a quality tracing process model and certificate registry data schema to achieve a smart contract implementation. Also, other blockchain applications can easily interact with the generated certificate registry smart contract since it follows ERC-721 standard. These results show that Lorikeet can provide efficient MDE support of
blockchain applications for non-fungible asset business processes and registries.

5.1.3. Task Outsourcing Process

Fig. 10 illustrates a task outsourcing business process model, which is an escrow use case for conditional payment. The process starts when a task requester lodges a task. The task requester selects a person from a list of matching workers and deposits the negotiated amount of money to the token registry. If the amount of deposited money is not correct or the recipient is wrong, the money is refunded back to the task requester. Otherwise, the money is released to the worker if the task is assessed as complete. The task outsourcing process model specifies the Lorikeet token registry using SmartContractInterface. On the registry side, we use the same fungible asset registry model for the Lorikeet token registry.

Lorikeet creates a smart contract for instantiating the task outsourcing process, taking the business process model as input while generating a token registry smart contract using the specified token information as fields and the ERC-20 standard methods as methods.
This use case shows that conditional payment (i.e. escrow) can be implemented using Lorikeet. Developers only need to model the payment process and token registry without writing Solidity code for complex escrow logic.

5.1.4. Grain Title Creation

Fig. 11 shows a simplified grain title creation process modelled using Lorikeet, which is focused on grain ownership transfer [20] and can be considered an asset swap use case. There are two asset registries interacting with this selected grain title creation process: grain title registry (non-fungible) and Lorikeet token registry (fungible).
The grain title creation process starts when a truck arrives and is weighed for the first time. The grain is dropped into a silo and the truck is weighed again to determine the net weight of grain that has been delivered (weight of truck before delivery minus weight of truck after delivery). Also, when the truck arrives, before dropping grain into the silo, a grain sample is taken and quality assessment is conducted. The process then creates the grain title and assigns it to the farmer, but puts it into escrow of the process. When the buyer pays the correct amount of money for the grain into the process escrow, process transfers the grain title to the buyer and the money to the farmer. If the amount is incorrect the money and the title are refunded to the buyer and the farmer, respectively.

The registry smart contract address and actions to be invoked for the grain title registry are specified as attributes of SmartContractInterface. On the registry side, we use the same fungible asset registry model for the Lorikeet token registry.

Lorikeet outputs a smart contract for the grain title creation process taking the business process model as input. For registry smart contracts, Lorikeet uses the specified token and grain title information to generate an ERC-20 compliant smart contract and an ERC-721 compliant smart contract, which can be easily integrated and communicated with other ERC smart contracts. The process of grain title creation involves various activities and interactions with both fungible asset registry and non-fungible asset registry.

When using Lorikeet, developers only need to model the process and the two asset registries without writing Solidity code for complex business logics, such as creating grain title, checking escrow balance, and asset swap. Also, the business process implemented as a smart contract enforces conformance of any execution with the grain title process, e.g., a grain title can only be created after grain weight is calculated and quality is evaluated.
5.2. Functional Correctness and Cost Analysis

We tested the functional correctness of Lorikeet by checking conformance of the generated business process, including interactions between the registry smart contracts and the business process smart contracts, with a test suite in the Given-When-Then structure [8]. We ran the experiment on Ganache, i.e. the blockchain client simulates an Ethereum blockchain, and compiled all the Solidity smart contracts using solc v.0.5.8 with optimization enabled.

In order to provide a concrete example, we used the grain title creation process as shown in Fig. 11. We started with deriving two permissible execution traces for the grain title creation process model, so-called conforming traces that adhere to the process model. One follows the successful Asset Swap path, while the other follows the failed Refund path. For each of these two traces, we generated 250 traces with randomized noise injected to obtain a larger set of traces (including both conforming and non-conforming traces) with the following manipulation operators: (i) add a new log line, (ii) remove a log line, or (iii) switch the order of two log lines, such that the modified trace was different from the two initial conforming traces. In total, there are 502 traces.

We investigated if our implementation accurately identifies the non-conforming traces that have been generated for the grain title creation process model. The results are shown in Table 2. All log traces were correctly classified. There are no assertion failures, which was our expectation. Any other outcome (i.e. assertion failure) would have pointed at severe issues with our approach or implementation. Thus, we claim if a trace is conforming, its execution will execute...
the correct logic that we expect. Our full test suite is available online\textsuperscript{13}.

We also investigated the cost of involving the blockchain in the process execution, since gas cost on Ethereum reflects computational effort and determines throughput on a given network. Each bytecode instruction in a smart contract consumes gas when it is executed in response to a transaction invoking the smart contract. The “size” of a block in Ethereum is specified as a block gas limit, i.e., the sum of gas consumed by all transactions in a block may not exceed this limit. As such, cost in gas affects all blockchain networks, regardless of whether they are public, private, or consortium blockchain networks.

Table 3 shows the average, maximum, and minimum value of gas used by each activity. Note that the variability in gas cost (difference between minimum and maximum) used by Grain\textsubscript{quality}\_evaluated and Truck\textsubscript{is\_weighed\_again} is large. The reason is that these two tasks are (eventually) followed by a parallel join gateway and thereafter a script task; only once the second of the two tasks completes, the gateway and subsequent script task are executed. Therefore, the cost depends on the execution order, resulting in the high variability. Also noteworthy is that Buyer\textsubscript{wants\_to\_buy\_title} is consistently expensive. It too is followed by script tasks and a gateway, but regardless of the gateway decision there are always two scripts executed. And all of the involved script tasks interact with the asset registries. Finally, both Failed and End are end events of the BPMN process, and the generated Solidity code for both involves re-setting a smart contract storage variable. This triggers a refund of 15,000 gas, which offsets the total gas consumption of the rest of the transaction, and hence results in a low total cost.

The average gas cost per transaction an instance of the process model is 59,497 gas (taking into account that only one of the outgoing branches of the XOR split is executed). For comparison, we used Google BigQuery’s Ethereum

\textsuperscript{13}https://drive.google.com/drive/folders/1gaWhCY2YK8m4MXUvoA69j2bNboJ8i7QZ?usp=sharing
Table 3: Gas used by each task in the grain title creation process

| Task                               | Avg   | Min   | Max   |
|------------------------------------|-------|-------|-------|
| Farmer wants to register grain title | 49491 | 49491 | 49491 |
| Truck carrying grain is weighed     | 49319 | 49319 | 49319 |
| Grain sample taken                  | 28983 | 28983 | 28983 |
| Grain dropped at silo               | 28938 | 28938 | 28938 |
| Grain quality evaluated             | 93055 | 49296 | 146411|
| Truck is weighed again              | 102321| 69892 | 182007|
| Buyer wants to buy title            | 109523| 104903| 119986|
| Failed                             | 14338 | 14338 | 14338 |
| End                                | 14349 | 14349 | 14349 |

dataset\textsuperscript{14} to determine the average cost of a transaction that invokes a smart contract, with the following query:

\begin{verbatim}
SELECT avg(receipt_gas_used)
FROM 'bigquery-public-data.ethereum_blockchain.transactions'
WHERE DATE(block_timestamp) <= "2019-12-11"
AND to_address is not null
AND to_address in
(SELECT address
FROM 'bigquery-public-data.ethereum_blockchain.contracts')
AND input != "0x"
\end{verbatim}

From this query, we obtained aggregate costs over 292,826,434 contract invocation transactions on the public Ethereum blockchain up until 11 December 2019: on average 86,330 gas; median of approx. 45,647.\textsuperscript{15} By comparison, transactions invoking functions of the smart contracts generated with Lorikeet use on average only 59,497 gas, or 68.9\% of the global average; the median is 49,405 gas.

\textsuperscript{14}https://cloud.google.com/blog/products/data-analytics/ethereum-bigquery-public-dataset-smart-contract-analytics

\textsuperscript{15}The median is approximate, since the large total number requires using approximate SQL functions like \texttt{APPROX_QUANTILES}.
gas, which is about 8% higher than the global median.

6. Conclusion and Future Work

Model-driven engineering (MDE) is of particular importance for blockchain-based applications since MDE tools can help developers focus on high-level modelling and generate well-tested smart contract code implementing best practices. A typical class decentralised applications uses blockchain to manage business processes that interact with asset registries, including processes for fungible/non-fungible asset registration, escrow for conditional payment, and asset swap. This paper tackled the non-trivial integration of business processes and asset management, which requires modelling support for various types of integrations, and smart contract generation based on the build models.

We proposed methods to specify models for business processes and asset registries, to interconnect them, and to generate smart contracts using the specified models. To support the proposed MDE approach, we designed and implemented a tool named Lorikeet. The proposed approach was evaluated in terms of feasibility and functional correctness using four industrial use cases. Code from the evaluation has been made available for reproducibility. The results show that developers can use our MDE approach as implemented in the Lorikeet tool to generate functionally correct smart contracts based on the business process and asset registry models.

A comparison of the gas consumption of a transaction to the generated smart contracts vs. over more than 292 million contract invocation transactions on public Ethereum shows that the former consume less gas on average, but at a slightly higher median. While this data is not suitable for drawing definitive conclusions from a comparison of absolute numbers – after all, we do not know what functions are implemented by other smart contracts – they give strong indication that Lorikeet smart contracts are not overly inefficient. Note that lower gas consumption corresponds to higher throughput and, on public blockchains, lower monetary cost – therefore gas consumption is relevant for private and
consortium blockchains as well.

Although we focus on the domain of business processes and asset management in this paper, our approach can be easily applied to a broad range of blockchain applications. In future work, we plan to extend our MDE approach to support more comprehensive access control policies across all types of generated smart contracts.

References

[1] Wil M. P. van der Aalst, Marlon Dumas, Chun Ouyang, Anne Rozinat, and Eric Verbeek. Conformance checking of service behavior. *ACM Trans. Internet Technol.*, 8(3):13:1–13:30, May 2008.

[2] Silvia Abrahão, Francis Bourdeleau, Betty Cheng, Sahar Kokaly, Richard Paige, Harald Stöerrle, and Jon Whittle. User experience for model-driven engineering: Challenges and future directions. In *MODELS’17*, pages 229–236, Sept 2017.

[3] Alexandra Bratanova, Dinesh Devaraj, Joanna Horton, Claire Naughtin, Ben Kloester, Kelly Trinh, Ingo Weber, and David Dawson. Blockchain 2030: A look at the future of blockchain in Australia. Technical report, Data61, CSIRO, Brisbane, Australia, April 2019.

[4] Barbara Carminati, Elena Ferrari, and Ngoc Hong Tran. Secure web service composition with untrusted broker. In *ICWS’14*, pages 137–144. IEEE, 2014.

[5] Sheng Chen, Zhengzhong Zhu, Jianguo Zheng, Zhenqiu Lu, Yanjun Huang, Meng Liang, and Chao Liu. Rules for quality tracing of import commodities (sn/t 4941-2017). Technical report, General Administration of Quality Supervision, Inspection and Quarantine of the People’s Republic of China, Beijing, China, 2017. (in Chinese).

[6] Paul Downey. The characteristics of a register, 2016. Accessed: 23 Feb 2018. [Online]. Available: https://gds.blog.gov.uk/2015/10/13/the-characteristics-of-a-register.

[7] Eric J. Evans. *Domain-Driven Design: Tackling Complexity In the Heart of Software*. Addison-Wesley, Boston, MA, USA, 2003.
[8] Ellen Gottesdiener. *Discover to deliver: agile product planning and analysis*. EBG Consulting, Incorporated, 2012.

[9] Bettina Kemme and Gustavo Alonso. Database replication: A tale of research across communities. *Proc. VLDB Endow.*, 3(1-2):5–12, September 2010.

[10] Guoli Li, Vinod Muthusamy, and Hans-Arno Jacobsen. A distributed service-oriented architecture for business process execution. *ACM TWEB*, 4(1):2:1–2:33, 2010.

[11] Orleny L´ opez-Pintado, Luciano Garcia-Bañuelos, Marlon Dumas, and Ingo Weber. Caterpillar: A blockchain-based business process management system. In *BPM’17*, pages 1–5, Barcelona, Spain, September 2017. Springer.

[12] Qinghua Lu, Ingo Weber, and Staples Mark. Why model-driven engineering fits the needs for blockchain application development. *IEEE Blockchain Technical Briefs*.

[13] Loi Luu, Duc-Hiep Chu, Hrishik Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS ’16, pages 254–269. ACM, 2016.

[14] Marco Casassa Mont and Lorenzo Tomasi. A distributed service, adaptive to trust assessment, based on peer-to-peer e-records replication and storage. In *FTDCS’01*, pages 89–95, 2001.

[15] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer electronic cash system, 2009. Accessed: 23 Feb 2020. [Online]. Available: http://www.bitcoin.org/bitcoin.pdf.

[16] Steve Omohundro. Cryptocurrencies, smart contracts, and artificial intelligence. *AI Matters*, 1(2):19–21, December 2014.

[17] Douglas C. Schmidt. Guest editor’s introduction: Model-driven engineering. *Computer*, 39(2):25–31, Feb 2006.

[18] David Siegel. Understanding the DAO attack, 2016. Accessed: 23 Feb 2020. [Online]. Available: http://www.coindesk.com/understanding-dao-hack-journalists.
[19] Anna Squicciarini, Federica Paci, and Elisa Bertino. Trust establishment in the formation of virtual organizations. In *ICDE'08 Workshops*, pages 454–461. IEEE Computer Society, 2008.

[20] Mark Staples, Shiping Chen, Sara Falamaki, Alex Ponomarev, Paul Rimba, An Binh Tran Ingo Weber, Xiwei Xu, and John Zhu. Risks and opportunities for systems using blockchain and smart contracts. Technical report, Data61 (CSIRO), Sydney, Australia, 2017.

[21] Sattanathan Subramanian, Philippe Thiran, Nanjangud C. Narendra, Ghita Kouadri Mostefaoui, and Zakaria Maamar. On the enhancement of BPEL engines for self-healing composite web services. In *SAINT'08*, pages 33–39, July 2008.

[22] An Binh Tran, Qinghua Lu, and Ingo Weber. Lorikeet: A model-driven engineering tool for blockchain-based business process execution and asset management. In *BPM Demo Track*, 2018.

[23] Binh Tran, Xiwei Xu, Ingo Weber, Mark Staples, and Paul Rimba. Regerator: a registry generator for blockchain. In *CAiSE'17, Forum Track (demo)*, pages 81–88, Essen, Germany, June 2017.

[24] Florian Tschorsch and Björn Scheuermann. Bitcoin and beyond: A technical survey on decentralized digital currencies. *IEEE Communications Surveys & Tutorials*, 18(3):2084–2123, 2016.

[25] Wattana Viriyasitavat and Andrew Martin. In the relation of workflow and trust characteristics, and requirements in service workflows. In *Informatics Engineering and Information Science*, pages 492–506. Springer, 2011.

[26] Ingo Weber, Xiwei Xu, Regis Riveret, Guido Governatori, Alexander Ponomarev, and Jan Mendling. Untrusted business process monitoring and execution using blockchain. In *BPM’16*, pages 329–347, Rio de Janeiro, Brazil, September 2016. Springer.

[27] Ingo Weber, Xiwei Xu, Regis Riveret, Guido Governatori, Alexander Ponomarev, and Jan Mendling. Using blockchain to enable untrusted business process mon-
itoring and execution. Technical report, University of New South Wales, 2016. Technical Report UNSW-CSE-TR-201609.

[28] M. V. Zelkowitz and D. R. Wallace. Experimental models for validating technology. *IEEE Computer*, 31(5):23–31, May 1998.

[29] Liangzhao Zeng, Boualem Benatallah, Anne H. H. Ngu, Marlon Dumas, Jayant Kalagnanam, and Henry Chang. QoS-aware middleware for web services composition. *TSE*, 30(5):311–327, May 2004.