future internet

MDPI

Article

Payment Rails in Smart Contract as a Service (SCaaS) Solutions from BPMN Models

Christian Gang Liu [1], Peter Bodorik [1,*] and Dawn Jutla [2]

[1] Faculty of Computer Science, Dalhousie University, Halifax, NS B3H 4R2, Canada; chris.liu@dal.ca
[2] Sobey School of Business, Saint Mary's University, Halifax, NS B3H 3C3, Canada; dawn.jutla@gmail.com
* Correspondence: peter.bodorik@dal.ca

Abstract

The adoption of blockchain-based smart contracts for the trading of goods and services promises greater transparency, automation, and trustlessness, but also raises challenges related to payment integration and modularity. While business analysts (BAs) can express business logic and control flow using BPMN and decision rules using DMN, payment tasks that involve concrete transfers (on-chain, off-chain, cross-chain, or hybrid) require careful implementation by developers due to platform-specific constraints and semantic richness. To address this separation of concerns, we introduce a methodology within the context of the smart contract-as-a-service (SCaaS) approach that supports (1) identifying and mapping generic payment tasks in BPMN to pre-deployed payment smart contracts, (2) augmenting BPMN models with matching payment fragments from a pattern repository, and (3) automatically transforming the augmented models into smart contracts that invoke the appropriate payment services. Our approach builds on prior work in automated BPMN-to-smart contract transformation using Discrete Event–Hierarchical State Machine (DE-HSM) multi-modal modeling to capture process semantics and nested transactions, while enabling payment service reuse, extensibility, and the separation of concerns. We illustrate this methodology via representative use cases spanning conventional, DeFi, and cross-chain payments, and discuss the implications for modular contract deployment and maintainability.

Keywords: payment rails; smart contract-as-a-service; BPMN; cross-chain payments; payment services repository; blockchain interoperability

Check for updates

Academic Editors: Shujie Yang, Xiaohai Dai, Zhuo Chen, Qinnan Zhang and Junqin Huang

Received: 18 January 2026
Revised: 7 February 2026
Accepted: 8 February 2026
Published: 19 February 2026

Copyright: © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

1. Introduction

Blockchains and their programmable smart contracts have gained significant interest due to their potential to automate agreements among distrusting parties without intermediaries, forming a cornerstone of Web3 ecosystems. Classic work on blockchain technology highlights both its trust guarantees and its challenges in real-world application domains involving multiple stakeholders and complex business logic. The development of robust, secure, and maintainable smart contracts remains a barrier to adoption, especially for business applications that include payment coordination across heterogeneous rails.

Writing smart contracts is difficult not only because of blockchain platform intricacies (e.g., gas, storage costs, and consensus), but also due to distributed coordination among participants—a longstanding challenge in distributed systems. For applications involving multiple actors, coordination, synchronization, and transactional guarantees increase development complexity significantly.

To reduce developer burden, the Model-Driven Engineering (MDE) paradigm has been applied whereby application requirements are first modeled at a high level of abstraction and then automatically transformed into executable artifacts. Early works used finite state machines (FSMs) to model contract logic (e.g., [1]) and Petri nets to capture concurrency and synchronization before contract synthesis. However, FSMs often lack the expressiveness needed for complex multi-actor processes, and Petri net models—though powerful—are not widely adopted by business practitioners due to their perceived complexity.

For this reason, many researchers have turned to Business Process Model and Notation (BPMN) for representing application requirements and control flow, because BPMN is understood by business analysts, managers, and software developers alike. BPMN naturally expresses conditional flows, parallelism, and collaboration, and it integrates with Decision Model and Notation (DMN) to capture business logic decisions (e.g., tariff choices, eligibility rules). DMN's Friendly Enough Expression Language (FEEL) allows business logic to be expressed in a readable and structured form that can be transformed into target implementation code.

Despite these advances, payment integration remains a challenge: it is unreasonable to expect a business analyst (**BA**) to specify low-level payment implementation details in the target programming language (e.g., Solidity on Ethereum). Different trade applications may require disparate payment methods (bank rails, crypto tokens, escrow, cross-chain bridges), each with distinct semantics and compliance constraints. Furthermore, modern systems increasingly hybridize on-chain and off-chain components, demanding orchestration logic that ensures atomicity and correct sequencing across disparate execution contexts.

Across all payment categories, two properties are especially relevant to the BPMN-to-smart contract transformation problem:

1. *Semantic richness*—Payments involve amounts, assets (currencies or tokens), participants, timing constraints, and compliance conditions, all of which must be respected by execution logic.
2. *Implementation diversity*—The actual execution semantics vary widely depending on the payment rail (traditional banking, DeFi protocols, escrow logic, wrapped assets, or cross-chain protocols), even though BAs typically model payments as abstract tasks.

This motivates the central abstraction of the smart contract-as-a-service (SCaaS) approach to payment services: BAs model payments generically, while developers produce concrete payment smart contract services that can be pre-deployed and invoked by an automatically generated orchestration of process execution from BPMN models.

In our previous work, we established a foundation for the automated generation of smart contracts from BPMN models using multi-modal modeling, comprised of discrete event (DE) and hierarchical state machine (HSM) modeling and referred to as DE-HSM modeling, that is used to capture process semantics, participant interactions, and execution ordering. We subsequently extended this pipeline to support nested and multi-step transactions. More recently, we introduced TABS+R, a mechanism that enables the repair and upgrade of generated smart contracts when real-world conditions diverge from modeled assumptions by isolating the affected BPMN fragments, modifying them, and regenerating consistent contract logic [2].

A key benefit of DE-HSM modeling within the SCaaS framework is its explicit separation of concerns. Collaborative interactions among participants, such as task sequencing and message exchanges, are modeled using discrete-event (DE) semantics, while the internal functionality of BPMN task elements is represented using Hierarchical State Machines (HSMs). Decision points in BPMN models are represented by gateways whose conditions are typically expressed as relatively simple Boolean expressions. These expressions, such as checking whether a payment amount exceeds a threshold value, can be naturally specified

by BAs using DMN and FEEL, which are designed to be accessible to both technical and non-technical stakeholders.

Our modeling approach supports a clear division of responsibilities: BAs focus on modeling trade activities and decision logic, while developers provide specialized services when the functionality of a BPMN task exceeds the BA's scope—most notably in the case of payment execution. However, existing BPMN-to-smart contract transformation approaches do not provide a systematic mechanism for abstracting payment rails, separating payment implementation from process logic, or reusing pre-deployed payment services across multiple trade applications.

1.1. Objectives

The focus of our project is to simplify the creation of smart contracts for blockchain applications that support the trade of goods and services. Our primary objective is to automate smart contract generation by allowing business analysts (BAs) to model trade workflows in BPMN, while software developers implement code for BPMN task elements as methods with well-defined inputs, outputs, and objectives.

The transformation process automatically synchronizes process flows and message exchanges among system components using DE-FSM modeling. This approach enables developers to focus on well-defined tasks without managing complex process synchronization, reducing errors and developer effort. It also supports a clear separation of concerns between BAs and developers, simplifying collaboration.

Within the context of generating smart contract-as-a-service solutions from BPMN models and the separation of concerns between the roles of the BAs and developers, this paper aims to:

1. Enable software developers to prepare specific smart-contract payment services in advance, targeting various payment rails (on-chain crypto, off-chain bank transfers, cross-chain rails).
2. Allow business analysts to model trade activities for goods and services in BPMN models while allowing them to use *generic representation of payment services* without requiring knowledge of payment implementation details.
3. Design a transformation process that identifies generic payment tasks in a BA's BPMN model, matches them to concrete payment services via a BPMN fragment repository, augments the BPMN model accordingly, and generates a smart contract that orchestrates the trade activity while invoking the appropriate payment services.

The scope of the paper concerns how smart contract payment services can be prepared and deployed while minimizing the development effort, rather than the efficiency of the produced software. Reducing the time required for a BA or a developer is more cost-effective than minimizing the execution delays of a smart contract method, unless that smart contract method is executed many times. The execution delays of a smart contract payment method depend on the method's efficiency and on the efficiency of the underlying infrastructure that executes the method. As the smart contract methods of a payment service are prepared by a software developer, the efficiency of those methods depends on the skill of the software developer and the underlying blockchain. Consequently, the efficiency of the payment services methods prepared by a software developer are outside scope of this paper.

1.2. Contributions

This paper extends the capabilities of our TABS+R system and makes the following contributions:

- *Separation of concerns between trade-activity modeling and payment implementation*, allowing BAs and developers to work independently at appropriate abstraction levels.
- *Repository-mediated transformation of generic payment tasks*, enabling BPMN payment elements prepared by BAs to be mapped to instantiated and deployed payment services prepared by developers.
- *Multi-rail payment support* for heterogeneous payment scenarios, including on-chain native payments, third-party cross-chain stablecoin payments, and off-chain conventional banking transfers.
- *Reuse and composition of patterns for extensibility of payment services*, enabling the same payment services to be leveraged across multiple trade applications.
- *Shared state management across participants* to facilitate sophisticated multi-party settlement patterns.
- *Cross-platform payment service invocation*, supporting interaction with heterogeneous blockchain and off-chain systems.
- *Incremental ecosystem growth*, allowing new payment services to be added to the repository without disrupting existing applications.

At the end of Sections 3 and 4, we summarize which of the objectives were reached, while in the last section we review the contributions listed above while also summarizing how they were achieved.

1.3. Outline

The remainder of this paper is organized as follows. Section 2 provides a background on conventional and crypto-payment methods, introduces BPMN and DMN modeling and their use in transformation pipelines, and overviews our SCaaS approach from BPMN and DMN models. Section 3 presents our approach to integrating payment services into SCaaS workflows, including repository design and smart contract generation, illustrated through a representative use case covering on-chain, off-chain, and cross-chain payments. Section 4 discusses reuse and extensibility by presenting an additional use case of creating a payment service that utilizes payments with settlement netting while reusing other, already existing, payment services and by showing how the repository supports the integration of services. Section 5 reviews related work, situating our contributions relative to existing BPMN-to-smart contract transformation research. Finally, Section 6 provides a summary, the contributions, and the conclusions with the direction of future work.

2. Background

This section surveys the foundational concepts underlying the generation of smart contracts from business process models and motivates the smart contract-as-a-service (SCaaS) paradigm with explicit support for payment rail abstraction. We first characterize payment methods across conventional and blockchain-based ecosystems, then introduce BPMN and DMN as modeling standards for business processes and decisions and finally review our approach to transforming BPMN models into executable smart contracts.

2.1. Payment Methods in Conventional and Blockchain Ecosystems

Payments represent the transfer of economic value between parties and differ substantially in semantics, execution guarantees, and compliance obligations depending on the underlying rail. These differences are especially relevant when payment logic is orchestrated by smart contracts generated from abstract process models.

2.1.1. Conventional (Off-Chain) Payments

Conventional payment systems operate over established financial infrastructures such as Automated Clearing House (ACH) transfers, wire payments, card networks, real-time gross settlement (RTGS) systems, and correspondent banking arrangements. These rails rely on intermediaries, including banks, clearing houses, and payment processors, to execute settlement, manage liquidity, and enforce regulatory requirements such as anti-money laundering (AML) and know-your-customer (KYC) checks. Payment instructions and confirmations are commonly exchanged using standardized messaging frameworks, notably SWIFT MT and MX formats [3,4].

Settlement in conventional systems is typically irreversible once completed, and dispute resolution is handled through institutional or legal processes external to the execution infrastructure. While compliance rules and exception-handling logic can be encoded in smart contracts, the determination of compliance and exception conditions typically depends on off-chain data, institutional authority, and human judgment. As a result, compliance evaluation and exception recognition cannot be performed natively on-chain and must be delegated to external services or oracles when integrating conventional payment rails with smart contracts [5].

2.1.2. Blockchain-Based Payments

Blockchain-based payment mechanisms enable programmable value transfer on distributed ledgers. These mechanisms can be classified based on ledger locality and the nature of inter-ledger interaction.

On-Mainchain Payments

On-mainchain payments occur when both the requesting smart contract and the payment execution logic reside on the same blockchain. Transfers of native cryptocurrencies or tokenized assets are atomic, transparent, and self-verifying due to blockchain consensus. Such payments support advanced features such as conditional transfers, escrow, and bulk payments with netting, all enforced directly by smart contract logic [6,7]. Regulatory compliance is typically layered via external oracles or application-level controls.

Cross-Chain Native Payments

Cross-chain native payments transfer value between distinct blockchains while preserving asset equivalence. Typical approaches rely on burning or locking assets on the source chain and minting or releasing corresponding assets on the destination chain via bridges or cryptographic interoperability protocols. Correct execution requires atomicity and state consistency across heterogeneous ledgers [8,9]. However, the most difficult issue is how to preserve asset equivalence, particularly if the asset value depends on its demand and may change rapidly, such as the value of publicly traded stocks or the prices of some of the crypto-coins.

Third-Party Cross-Ledger Payments

In third-party cross-ledger payments, a trade activity smart contract deployed on one blockchain invokes a payment service hosted on a different blockchain. The 3rd-party payment service executes the transfer using its native asset, such as the crypto-coin stablecoins USDT or USDC, or an emerging Central Bank Digital Currency (CBDC) and returns a success or failure response. As the payment service is based on a payment request/reply message exchange, this interaction resembles and is functionally similar to off-chain payment services from the caller's perspective [10,11].

Clearly, 3rd-party payment services are a special subset of cross-chain native payments, in which both the payer and payee have accounts on the 3rd-party payment service blockchain, and both parties agree on a specific payment value expressed in the native crypto-coin of the payment service blockchain. It is expected that when CBDC payment services are established, they will become the dominant payment service that is used the most often and will become the corner stone of Web3 payment services.

Hybrid Payments

Hybrid payment systems combine off-chain, on-mainchain, and cross-chain components using wrapped assets, bridges, or interoperability frameworks. While offering flexibility, these systems introduce additional orchestration complexity, making them a natural fit for programmable controllers implemented as smart contracts [12].

Across all payment categories, two properties are particularly salient for BPMN-to-smart contract transformation: semantic richness, encompassing amount, asset type, timing constraints, participating parties, and compliance conditions; and implementation diversity, whereby identical business-level payment tasks map to radically different execution semantics depending on the rail [13,14]. These characteristics motivate a service-oriented abstraction in SCaaS, where business analysts model payments generically and developers implement reusable payment services.

2.2. BPMN, DMN, and FEEL

Model-driven smart contract generation depends on modeling standards that balance expressiveness with accessibility for business stakeholders.

2.2.1. Business Process Model and Notation (BPMN)

Business Process Model and Notation (BPMN) is an ISO-standard graphical language for specifying business processes involving tasks, events, gateways, message flows, and collaborations [15]. BPMN enables business analysts to express control flow, parallelism, and participant interaction in an intuitive manner. Despite its rich expressiveness in terms of process-flow modeling, it was designed to act as a "common language" that bridges the communication gap between initial business process design and final technical implementation. Thus, it was specifically designed to be understood by three primary groups of business stakeholders, namely: business analysts who use the notation to create the initial drafts and refinements of the business processes; technical developers who are responsible for implementing the technology and software components that will execute those processes; and business managers who use the diagrams to monitor, manage, and make strategic decisions based on the processes.

Importantly, BPMN is technology-agnostic: tasks describe business intent without embedding execution-level semantics, which facilitates transformation into multiple target platforms but requires additional interpretation during code generation.

2.2.2. Decision Model and Notation (DMN) and FEEL

Decision Model and Notation (DMN) complements BPMN by providing a standardized approach for modeling business decisions and rules [16]. DMN decision logic is expressed using the Friendly Enough Expression Language (FEEL), a declarative, human-readable language designed to be both understandable by business users and executable by machines. BPMN tasks may invoke DMN decisions to guide control flow, enabling separation between process structure and decision logic in model-driven transformations [15,16].

2.3. BPMN-to-Smart Contract Transformations

A substantial body of research explores transforming BPMN models into executable artifacts, including smart contracts. These approaches commonly treat BPMN as a high-level specification of process behavior, which is mapped to executable representations such as finite state machines or choreographed services [17–19]. In the following, we shall briefly overview our approach to transforming BPMN models into smart contracts, an approach which we refer to as SCaaS, and its tool, the SCaaS tool, which we called in our previous papers as the TABS tool, later TABS+, and TABS+R, as the tool evolved [20]. In a later section, called Related Work, we situate our work within the context of research on the transformation of BPMN models into smart contracts. Our primary differentiator from other work on this topic is that we have developed a multi-modal modeling approach that employs Discrete Event–Hierarchical State Machine (DE-HSM) representations to bridge BPMN models and smart contracts.

Our approach and the tool created convert BPMN models into smart contracts through an intermediate DE-HSM representation that captures both control flow and execution state. Logic-Structured Independent (LSI) subgraphs of BPMN models are identified and transformed into executable smart contract methods, enabling modular generation and optional sidechain execution for cost and privacy optimization. We further extended the framework to support multi-step collaborative transactions involving multiple participants. By identifying the transactional scopes within BPMN models, the tool generates smart contract logic that enforces transactional properties across actors.

Another extension to our approach and the tool we created addresses the need to handle the unanticipated events that arise in trade. While automated BPMN-to-smart contract generation and nested transaction address some of the design–time challenges, real-world processes inevitably evolve or encounter unanticipated runtime conditions. To improve usability and robustness, we extended the transformation pipeline in two key directions:

(i) First, business analyst-driven logic is enabled through integration with Decision Model and Notation (DMN). Business analysts can specify conditional decision logic using DMN and FEEL directly within BPMN models, and this logic is automatically incorporated into the generated smart contracts. As a result, domain experts can modify process behavior and decision rules without direct involvement from smart contract developers, significantly improving accessibility and maintainability.

(ii) Second, repair and upgrade support is provided to address runtime failures and process evolution. The repair identifies the innermost failing transactional BPMN fragment, allowing the modeler to revise the corresponding BPMN/DMN logic and regenerate the smart contract. The regeneration process preserves execution consistency by systematically reconstructing contract logic and transaction mechanisms, enabling controlled upgrades while maintaining state continuity [20].

Together, these extensions make the BPMN-to-smart contract pipeline resilient to change and failure, supporting end-to-end automation from modeling through to deployment, repair, and evolution in practical business settings. Payment tasks, however, often involve external services, heterogeneous rails, and reusable protocols. These limitations motivate the SCaaS abstraction, which separates payment implementation from process logic and enables payment services to be invoked as modular components during smart contract generation.

3. Making Payments—Service Modeling, Deployment, and Use

When considering payment services, there are two perspectives, payer and payee. To receive a payment, the payee needs to provide the information needed to make the payment using a specific payment method. For instance, the instructions may state that

the payment is to be made to a specific bank account, while also providing information that depends on whether the transfer is an inter- or intra-bank transfer and whether it is a cross-border payment. For a crypto-payment method, the payee needs to identify the crypto token/cryptocurrency and the other information required, such as an invoice or payment ID that identifies the goods/services for which the payment is being made.

We first overview, in Section 3.1, the overall process of making payments, how they are modeled, stored, and deployed within the SCaaS pipeline, and how generic payment tasks in business process models are systematically mapped to specific pre-deployed payment smart contracts. Next, we detail how payment service smart contracts are prepared and deployed (Section 3.2). We then present the structure and use of a payment service repository, which mediates the mapping between generic BPMN payment tasks and concrete services (Section 3.3). Finally, we describe the end-to-end transformation process that generates trade smart contracts from augmented BPMN models while invoking repository-selected payment services (Section 3.4). This exposition describes how the interpretive transformation framework, previously established for BPMN-to-smart contract synthesis using DE-HSM multi-modal modeling, was extended to support payment services. We conclude with a brief discission on the achievement of our objectives.

3.1. Overview of Payment Services in SCaaS

Our approach extends the SCaaS paradigm by decoupling business-level payment modeling from payment implementation, enabling business analysts (BAs) to express payments generically while developers supply executable services. In SCaaS, BPMN models are transformed into Discrete Event–Hierarchical State Machine (DE-HSM) models that preserve BPMN control flow semantics and facilitate automatic smart contract generation. Generic payment actions in BPMN are modeled as simple task elements annotated with necessary data objects, such as payee, invoice ID, amount, currency, and other domain-specific parameters. These annotations capture the semantic inputs flowing into the payment task, providing the metadata needed to later select the appropriate payment services.

Unlike task elements that represent on-chain logic or business interactions (captured via BPMN tasks and DMN decision rules), a generic payment task does not itself encode execution semantics. Instead, at transformation time, the SCaaS tool consults a payment service repository to identify which concrete payment smart contract (pre-deployed by developers) matches the semantic characteristics of the generic task. Once selected, the BPMN model is augmented with the BPMN fragment corresponding to that payment service and transformed into a composite smart contract that invokes the selected service rather than reimplementing payment logic. This separation supports reuse, modular deployment, and extensibility.

3.2. Preparation and Deployment of Payment Service Smart Contracts

Payment service smart contracts, whether payables or receivables, are developed by software engineers as reusable SCaaS services targeting specific payment rails (e.g., conventional banking via off-chain APIs, on-chain native token transfers, or cross-chain services via bridges/oracles). Each service originates from a payment BPMN fragment model that describes the service's control and data flow, including the required inputs and outputs.

The developer first prepares the BPMN fragment to model the payment. Following the standard SCaaS transformation, the BPMN fragment representing the payment is processed into an equivalent DE-HSM model and then automatically compiled into smart contract methods. Unlike the BPMN-to-smart contract transformation approaches of other researchers, in our approach the concurrency and control flow are represented

via DE queues, and state machines preserve semantic correspondence between BPMN and contract logic. Once synthesized, the payment service contracts are deployed to designated blockchain environments (e.g., Ethereum, a DeFi-capable chain, or a sidechain configured for cross-chain interactions). Deployment metadata, including the method signatures, required parameters, compliance constraints, and supported rails, is recorded in the payment repository for later retrieval.

Conceptually, payment services fall into the following categories based on their execution context. The first classification is whether the payment services are for (a) making payments or for (b) receiving payments, which are also referred to as payables and receivables, respectively. Both payable and receivable services are further classified as follows:

i. *On-chain native payments*—Simple transfers of native tokens (e.g., ETH) between accounts on the same blockchain. These payments are atomic and synchronous, and smart contract invocation typically triggers immediate settlement within the same transaction context.
ii. *Cross-chain crypto-payments*—Payments facilitated across distinct blockchains using bridges or oracle mechanisms. These involve auxiliary/helper contracts on each chain to manage requests and callbacks, resulting in asynchronous execution and eventual consistency once the cross-chain protocol finalizes.
iii. *Off-chain conventional payments*—Payments mediated by external financial systems (e.g., bank transfers) via HTTP APIs or web services. These are inherently asynchronous and non-atomic, requiring a two-phase interaction pattern (request and callback) to reconcile success or failure.

Developers must ensure that each deployed payment service complies with the required protocols, including cross-chain communication interfaces or third-party API integration for off-chain rails.

3.3. Payment Service Repository

To mediate between generic BPMN payment tasks and specific deployed services, we maintain a structured *payment service repository*. Each entry in the repository corresponds to a deployed payment contract and includes information describing the smart contracts and their methods [6,21,22].

- Semantic descriptors—Formal metadata capturing payment type (payable/receivable, on-chain, off-chain, cross-chain), expected currency or token, blockchain network identifiers, and compliance requirements.
- Method signatures—Information about the required method parameters (e.g., sender/receiver identifiers, amounts, currency codes), event callbacks, and expected outputs.

Repository entries are indexed to support structured search based on BPMN task metadata, with the first index determining whether the service is for making or receiving a payment. When the SCaaS tool encounters a generic payment task during transformation, it uses the task's annotated inputs to filter candidate services that satisfy the semantic criteria (e.g., payable vs. receivable, correct currency, supported rail, required data fields). Partial matches prompt interactive refinement, and multiple matches allow user selection among alternatives. Once resolved, the system links the BPMN payment task to the selected service's BPMN fragment and embeds invocation logic in the augmented model.

This repository mechanism supports repository evolution because new services can be inserted when deployed, and usage logs can be stored to identify commonly selected payment rails for particular business contexts.

3.4. Transforming the Augmented BPMN Model into Smart Contracts

Once payment services are selected and integrated into the BPMN model, the SCaaS transformation pipeline proceeds as follows:

1. *BPMN augmentation*: Generic payment task elements are replaced or extended with repository-selected BPMN fragments that represent concrete payment service invocation logic, preserving original sequencing and exception handling.
2. *DE-HSM conversion*: The augmented BPMN model—now containing both business logic and payment semantics—is transformed to a DE-HSM multi-modal model. The DE layer captures event sequencing and concurrency, while HSM sub-models encode task execution semantics (both business and payment).
3. *Smart contract synthesis*: Using the TABS transformation mechanism, each DE-HSM sub-model is translated into smart contract methods. Payment service invocation is realized by embedding calls to the deployed payment smart contracts (identified via repository metadata), using appropriate inter-contract communication patterns and callback interfaces where necessary.
4. *Deployment and runtime artifacts*: The generated smart contracts are deployed to the target blockchain(s), and auxiliary APIs are produced to interface with the application layer. Integration with bridge/oracle services (for cross-chain rails) and off-chain callback handlers (for conventional payments) is managed via event listeners and workflow coordination logic.

Integrating payment services at transformation time preserves the developer's prior work and ensures that payments occur according to pre-tested semantics, rather than regenerating payment code for each trade process. This enhances modularity, reduces redundancy, and aligns with the SCaaS objective of factorizing reusable service logic from business process orchestration. In the following subsection, we describe the transformation process in a use case.

It should be noted that in the rest of the paper, when we refer to the payment services, we refer to both payable and receivable services. When we wish to differentiate, we shall make explicit references to payment-making services or payment-receiving services, or to payable or receivable services, respectively.

Also note that the repository stores the BPMN fragment used for transformation in a payment service smart contract in addition to the other information, such as whether it is a payable or receivable service, whether off-chain or some other type of service. The BPMN fragment model is included as it is necessary for interpreting the status events emitted by smart contracts. The following subsection will provide examples to clarify.

3.5. End-to-End Process for the Use Case

To show the end-to-end process for generating a blockchain application from a BPMN model that includes payments, we first describe a use case utilized for exposition purposes and overview our experimentation environment. We then describe the creation of payment service smart contracts by the developer and overview the process of transforming the BPMN model of the trade activity into smart contracts that invoke the payment services prepared by the software developer. Finally, we describe the execution and monitoring of the smart contract execution.

3.5.1. Sample Use Case

The BPMN model for the sample use case is shown in Figure 1. It shows a BPMN model that represents a sale of a large product, such as a combine harvester. Modeling is carried out by a business analyst (BA) who is assumed to be proficient in BPMN and DMN modeling, including the use of the FEEL for decision logic. Additionally, we assume that

the BA is familiar with JavaScript Object Notation (JSON), which is used to describe the flow of information throughout the computation process, as will be detailed later.

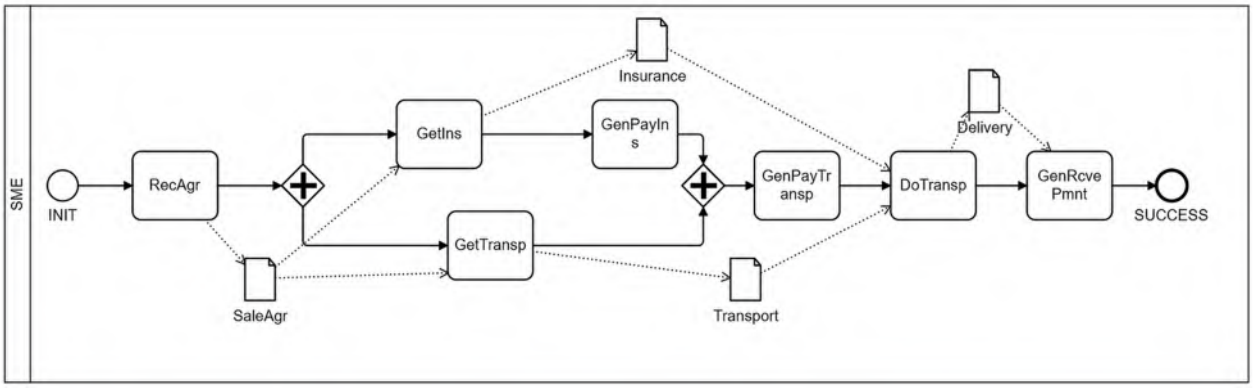


Figure 1. Sample use case: BPMN model with generic payments *GenPayIns*, *GenPayTransp*, and *GenRcvePmnt*.

In Figure 1, the first task, *RecAgr*, involves receiving an already-approved purchase agreement, such as an approved purchase order. The agreement is represented by a BPMN-associated data element, *SalesAgr*. The BA prepares the requests for insurance and transport represented by the task elements *GetIns* and *GetTransp*, while providing the task element information via the associated document *SalesAgr*. The task *GetIns* obtains the insurance contract, represented by the associated *Insurance* document, which is followed by the task element *GenPayIns* representing the generic payment for the insurance that was obtained.

The task *GetTransp* obtains a contract from a transporter for the product transport, with the contract being described in the associated *Transport* document. Obtaining and paying for insurance and obtaining the transporter can be executed concurrently, as indicated by the split parallel gateway (diamond shape with a plus sign). Once both the insurance and transport contracts are obtained, the payment to the transporter is made, as represented by the generic payment task *GenPayTrasnp*. The product is then received by the buyer, which is recorded in the associated *Delivery* document, which is forwarded to the final task, *GenRcvePmnt*, that receives the payment from the buyer.

It should be noted that this use case was utilized as a sample use case in our previous publications [23,24]. In [24], we described how a BA, working within the context of an SME, creates a BPMN model to track activities, document flows, and express the business logic decisions of BPMN task elements using DMN modeling. However, as we are dealing with payments, we extended this use case for the purposes of this paper with the following payments:

- Before the insurance can take place, it must be paid, which is represented by the BPMN task element *GenPayIns*.
- Payment for the transport is performed once the transport is completed. The payment is represented by the *GenPayTransp* task element.
- Once the product is received, the buyer pays for the product, which is represented by the *GenRcvePmnt* BPMN task element.

Furthermore, the naming, or labeling, of the BPMN task elements is such that any task name that starts with the substring "*GenPay*" represents a payable payment service, wherein the rest of the string name must ensure the name's uniqueness. Similarly, any BPMN task element with a name/label that starts with a substring "*GenRcve*" represents a general receivable payment service.

3.5.2. Execution Environment

We use three cloud servers provided by DigitalOcean [25]. Each server is equipped with two CPUs, 4 GB of memory, and 80 GB of disk space, and runs the Ubuntu operating system. Ganache-CLI was installed, configured, and executed on each server with different parameters to emulate a realistic deployment environment. The mainchain blockchain was configured with varying parameters for the block time, number of endorsers, and block size, using values close to those of the public Ethereum blockchain. The sidechain networks were configured using Quorum for Ethereum smart contracts and Microfab for Hyperledger chaincode.

In addition, the TABS tool allows testers to deploy smart contracts on the public Ethereum mainnet and the Ropsten test network, provided that sufficient cryptocurrency is available on those blockchains. Furthermore, each server also functions as an IPFS node within a preconfigured private IPFS cluster, ensuring that each participant has dedicated access to its corresponding IPFS node and storage space.

The evaluation process for a use case begins with BPMN modeling. Upon completion of the modeling phase, the SCaaS tool automatically transforms the BPMN model into a DE-HSM model during the design phase. The modeler then provides implementation code for the tasks defined in the BPMN elements, including the selection of the deployed services for the generic payment tasks present in the BPMN model.

After the compiled smart contracts are deployed on the blockchain networks, the modeler selects the evaluation mode to be used during the runtime phase. During execution, the modeler can observe the invocation of the smart contract methods. The SCaaS tool supports evaluation of the deployed model using an input stream that simulates the expected inputs from a decentralized application (DApp), as they would be submitted through the DApp interface.

Performance metrics collected during execution include the delay per smart contract invocation, the completion time of the individual DE-HSM sub-models, and the total execution time of the DApp, measured from the start event to the end event in the BPMN diagram. These metrics are intended primarily for the diagnostic and comparative evaluation of the execution behavior rather than for absolute performance benchmarking.

From the perspective of BPMN semantics, once the flow of control arrives at a BPMN task element representing a payment service, the task completes its execution before control leaves the task element via an outgoing sequence flow. This semantic property applies uniformly to on-mainchain, cross-chain, and off-chain payments. In cases where payment execution is asynchronous at the infrastructure level (e.g., cross-chain or off-chain payments), the SCaaS-generated orchestration enforces the BPMN task completion semantics by blocking control-flow advancement until a verifiable success or failure attestation is received from the corresponding payment service.

The SCaaS tool is configured primarily for research, experimentation, and testing. Accordingly, it includes features and processing steps not intended for production deployment, such as the ability to step through individual transformation phases and inspect intermediate models and artifacts. The tool also supports step-by-step execution for manual verification, as well as bulk issuance of transactions to measure average execution delays under varying workloads.

While the tool orchestrates execution and monitors emitted events, it does not participate in blockchain consensus or affect the correctness of smart contract execution, which remains entirely enforced by the underlying blockchain platforms. Captured events, emitted during the execution of smart contract methods, are used to form an execution trace used to verify the correctness of the smart contracts generated from the BPMN models.

3.5.3. Payment Services and Repository

When an invoice is submitted for goods or services, it includes payment instructions that specify the required execution details. If the payment repository does not already contain a corresponding service, a software developer must create one based on the payee's instructions.

The payment repository consists of two components:

1. *Generic payment methods*: Contain descriptions of various payment methods and identify which BPMN models utilize them.
2. *Deployed payment services*: Store information on active deployed services and identify which of the generic counterparts use them.

As described in Section 3.2, the developer first models the BPMN fragment that represents the payment service and then uses the SCaaS tool to transform the model into the methods of a smart contract and the APIs. We assume that instructions for the insurance payment, represented by the BPMN task element *GenPayIns*, indicate that the 3rd-party crypto-payment is to be made using the *USDXCrypto-StCn* stablecoin blockchain, while the payment instructions for the *GenPayTransp* indicate that the payment is to be made via an off-chain bank transfer. Finally, the task element *GenRcvePmnt* represents the payment-receiving service, which is to be performed by the on-mainchain crypto-payment, i.e., it is to be made by the crypto-coin of the mainchain on which the trading activity is deployed, and hence the payment service and the trade activity smart contracts share the ledger.

Here we shall concentrate on the basic payment services accomplished by a payment service that does not reuse other payment services. The payment methods of the sample use case are created by the software developer, but once created, they can be re-used as shall be demonstrated in the next section.

From the perspective of the BPMN semantics, once the flow of control arrives at the BPMN task element representing a payment service, it completes its execution before the flow of control leaves the task element via an exit sequence flow represented by an outgoing arrow from the task element. These BPMN semantics apply to all payments, regardless of whether the payment is on the mainchain, cross-chain, or off-chain. However, from an execution architecture point-of-view, there are differences depending on the implementation of the payment services.

If the payment services and the trade activity share the ledger, then the payment is accomplished by an atomic transfer of crypto units. However, the cross-chain and off-chain payments require the use of a helper smart contract or an oracle to provide a bridge to the crypto-payment blockchain: the helper smart contract method interacts with the target crypto-payment blockchain or the off-chain service requesting it makes the payment and then receives an indication from the payment service of the success/failure.

The role of this helper contract is not to execute the payment logic, but rather to (i) register the payment intent, (ii) record attestations of the payment outcomes, and (iii) emit verifiable events that allow the on-chain execution to progress. The use of a helper contract provides a uniform abstraction for off-chain and cross-chain payment services, including crypto-based payment rails (e.g., sidechain transfers, cross-chain payments) and conventional payment services (e.g., fiat payment processors). In this unified model, the main smart contract interacts exclusively with the helper contract, while the helper contract interfaces, directly or indirectly, with off-chain services.

However, for purely conventional off-chain payment methods, where no blockchain-resident artifact is required beyond the confirmation of payment completion, the helper contract may be omitted. In such cases, the off-chain payment service interacts with the main smart contract through a minimal callback mechanism that submits an attestation of payment completion. Conceptually, this corresponds to a degenerate instance of the

helper–contract pattern. However, we do not support such an option. For further details on helper smart contracts, please see the Appendix A.1.

3.5.4. Transformation of the BPMN Model and Execution

The SCaaS tool utilizes the Camunda platform [26] for creating BPMN models in accordance with the BPMN specifications [15,16] and stores the created BPMN models in XML format. Once a BPMN model is saved in XML form, it is transformed through a sequence of steps. The tool supports the generation of smart contracts for both Hyperledger Fabric blockchains and Ethereum Virtual Machine (EVM)-based blockchains, such as Quorum and Ethereum. The generated mainchain smart contract can invoke the methods of smart contracts deployed on a sidechain or use helper smart contracts to invoke the methods of smart contracts on other blockchains. The Appendix A.2 provides further details on the UI for the SCaaS tool using screenshots.

It should be noted that the SCaaS tool is configured primarily for research, experimentation, and testing. Accordingly, it includes features and processing steps that are not intended for production environments, such as the ability to step through individual transformation phases and inspect intermediate inputs and outputs. The tool also supports the issuance and execution of multiple transactions and enables the measurement of execution delays by capturing timestamps at various points during processing.

After the BPMN model in Figure 1 has been prepared by the BA using the tool and the transformation process has been initiated, the tool converts the BPMN model into a directed-graph representation of a DE-HSM model and analyzes it to identify fragments suitable for treatment as multi-step transactions and then the following steps deal with the transformation of generic payments into invocations of payment services prepared by the software developer.

For each generic payment task, the tool presents the available payment services and prompts the modeler to select which of one of them should be used to realize/instantiate the generic payment task. In the sample use case, there are three generic payment tasks: one receivable and two payable payments. Figure 2 illustrates the selection process. In the right-hand pane, the modeler first selects a generic payment task and then selects the corresponding payment service to be used. The figure shows that there are three generic payment services listed horizontally, *GenPayIns*, *GenPayTransp*, and *GenRcvePmnt*, and that the user has chosen the *GenRcvdPmnt* service (the button next to it has been highlighted) for selection of the payment service that is to be deployed for the generic service. Below the horizontal list of the generic payment services, there is the list of payment services, developed and deployed by the software developer, shown in a vertical column style, with each payment service having a selection button next to it. The listed payment services are *RcveCrypto-MnCh*, *USDXCrypto-paySt*Cn, *USDYCrypto-paySt*Cn, and *BankX-payOffCh*. Furthermore, the figure also shows that for the chosen *GenRcvdPmnt* service, the modeler has selected the *RcveCrypto-MnCh* payment service.

As shown on the diagram in a vertical list of available payment services, the following services were implemented and deployed:

- *RcveCrypto-MnCh* . . . On-mainchain payment-receivable service.
- *PayUSDXCrypto-StCn* . . . Third-party payable service on the *USDXCrypto-StCn* stablecoin blockchain.
- *PayUSDYCrypto-StCn* . . . Third-party payable service on the *USDYCrypto-StCn* stablecoin blockchain.
- *PayBankX-OffCh* . . . Off-chain bank receivable service.

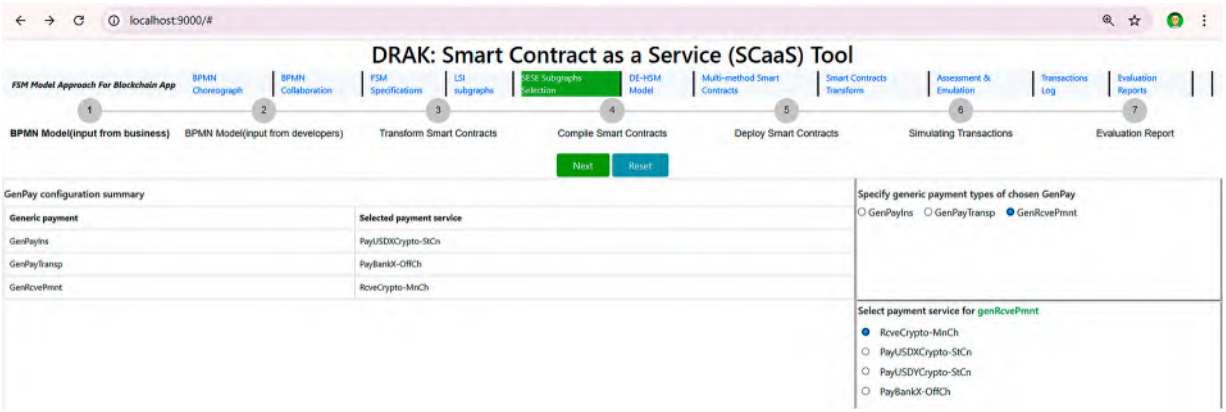


Figure 2. SCaaS tool showing selection of payment services for generic payment tasks.

The listed payment services are doubles we developed for testing, i.e., they are properly invoked; however, implemented payable services are mocks that are used to test and verify the behavior and interactions between the trade activity and payments.

To the left of the list of generic services and the payment services available, in the left-hand side pane, there is a list of the selections made for the generic services. It shows that the following selections were made by the user:

- *GenPayIns* . . . *USDXCrypto-StCn*—3rd-party payable service via the *USDXCrypto-StCn* blockchain.
- *GenPayTransp* . . . *BankX-OffC*—Off-chain bank receivable service.
- *GenRcvePmnt* . . . *RcveCrypto-MnCh* —On-mainchain payment-receivable service.

For each selected payment service, the SCaaS tool verifies that all the required information flowing into the BPMN task element is present. If required data is missing, the tool notifies the modeler, who must either amend the BPMN model to include the missing information or select a different payment service. If none of the existing services are applicable, a new payment service must be created by a software developer.

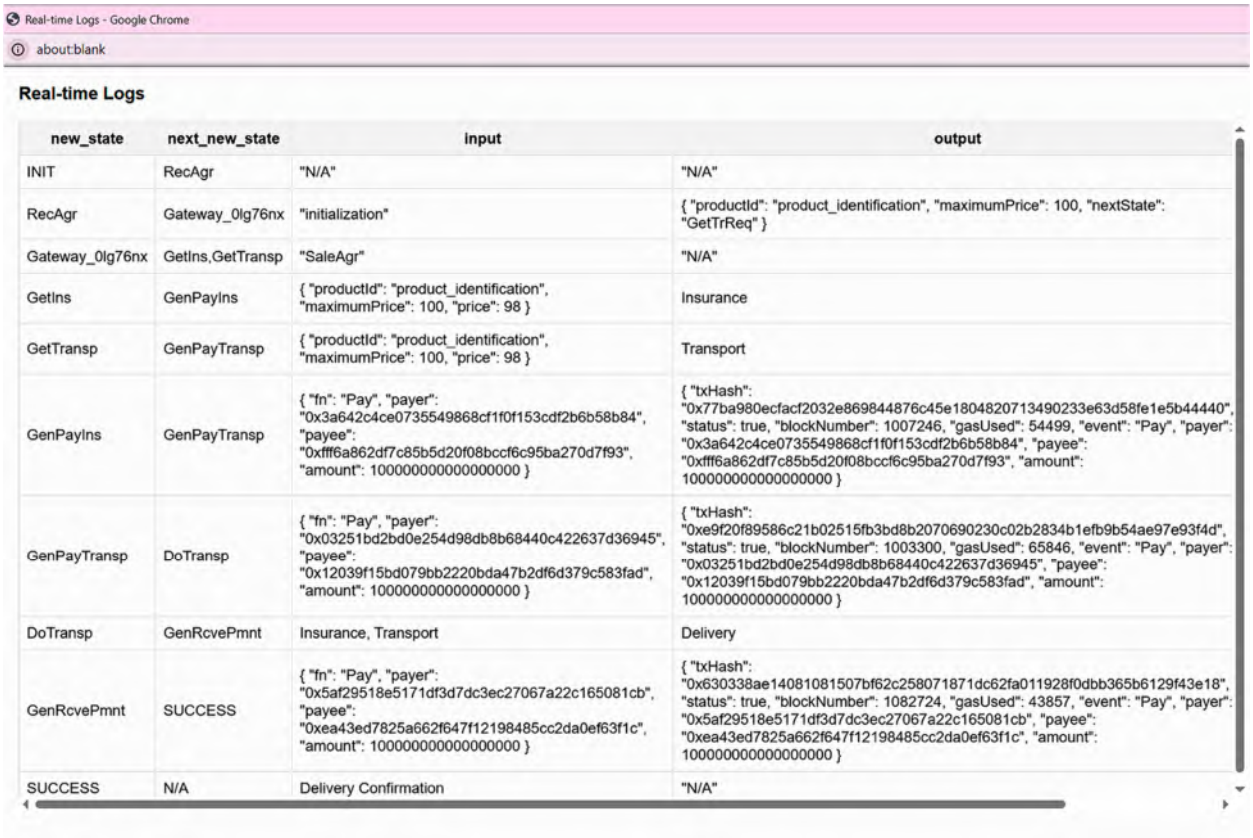
After the generic payment services are mapped to the payment services to be used, the tool performs the transformation of the BPMN model into the smart contract methods and API, and the trade activity smart contract is deployed on the target blockchain. In addition, the smart contract is deployed on the target blockchain.

Our smart contract methods emit event messages, which are collected by a dedicated API, for monitoring the state of execution of smart contracts. The event messages provide detailed information about changes in the execution state and, in addition to monitoring the state of smart contract execution, they are used for multiple other purposes, including manual verification, debugging, and the collection of performance metrics (e.g., execution delays) for evaluation. In general, the event messages convey the current execution status and maintain a log of execution progress.

Our long-term plan is to use these messages to display the status of execution of the trade activity. Specifically, the messages will enable reverse mapping from the execution model back to the BPMN model that will be presented to the operator in graphical form. In this visualization, BPMN elements that have been successfully executed will be shown in green, currently executing tasks in orange or yellow, and tasks that have not yet been executed in gray. Failed activities will be highlighted in red.

At present, the event messages emitted by the smart contracts are displayed in a separate window as they are received. Figure 3 shows this event-message window in which event messages are organized into four columns. The first column shows the name of the executed task or BPMN element. The second column lists the tasks that may be executed

next upon completion of the current task. The third column displays the parameters used by the task, including the data that flows into the corresponding BPMN task element via incoming sequence flows. The fourth column presents the task result, i.e., the content of the transaction as recorded on the blockchain used for the payment service.


Real-time Logs - Google Chrome

about:blank

Real-time Logs

new_state	next_new_state	input	output
INIT	RecAgr	"N/A"	"N/A"
RecAgr	Gateway_0lg76nx	"initialization"	{ "productId": "product_identification", "maximumPrice": 100, "nextState": "GetTrReq" }
Gateway_0lg76nx	GetIns,GetTransp	"SaleAgr"	"N/A"
GetIns	GenPayIns	{ "productId": "product_identification", "maximumPrice": 100, "price": 98 }	Insurance
GetTransp	GenPayTransp	{ "productId": "product_identification", "maximumPrice": 100, "price": 98 }	Transport
GenPayIns	GenPayTransp	{ "fn": "Pay", "payer": "0x3a642c4ce0735549868cf1f0f153cdf2b6b58b84", "payee": "0xfff6a862df7c85b5d20f08bccf6c95ba270d7f93", "amount": 100000000000000000 }	{ "txHash": "0x77ba980ecfacf2032e869844876c45e1804820713490233e63d58fe1e5b44440", "status": true, "blockNumber": 1007246, "gasUsed": 54499, "event": "Pay", "payer": "0x3a642c4ce0735549868cf1f0f153cdf2b6b58b84", "payee": "0xfff6a862df7c85b5d20f08bccf6c95ba270d7f93", "amount": 100000000000000000 }
GenPayTransp	DoTransp	{ "fn": "Pay", "payer": "0x03251bd2bd0e254d98db8b68440c422637d36945", "payee": "0x12039f15bd079bb2220bda47b2df6d379c583fad", "amount": 100000000000000000 }	{ "txHash": "0xe9f20f89586c21b02515fb3bd8b2070690230c02b2834b1efb9b54ae97e93f4d", "status": true, "blockNumber": 1003300, "gasUsed": 65846, "event": "Pay", "payer": "0x03251bd2bd0e254d98db8b68440c422637d36945", "payee": "0x12039f15bd079bb2220bda47b2df6d379c583fad", "amount": 100000000000000000 }
DoTransp	GenRcvePmnt	Insurance, Transport	Delivery
GenRcvePmnt	SUCCESS	{ "fn": "Pay", "payer": "0x5af29518e5171df3d7dc3ec27067a22c165081cb", "payee": "0xea43ed7825a662f647f12198485cc2da0ef63f1c", "amount": 100000000000000000 }	{ "txHash": "0x630338ae14081081507bf62c258071871dc62fa011928f0dbb365b6129f43e18", "status": true, "blockNumber": 1082724, "gasUsed": 43857, "event": "Pay", "payer": "0x5af29518e5171df3d7dc3ec27067a22c165081cb", "payee": "0xea43ed7825a662f647f12198485cc2da0ef63f1c", "amount": 100000000000000000 }
SUCCESS	N/A	Delivery Confirmation	"N/A"

Figure 3. Window with event messages showing execution trace.

In summary, for evaluation purposes, we utilize use cases, while verifying correctness through execution traces.

The figure captures the flow of control in the BPMN model as it occurred during execution of our sample use case. The event messages indicate that the execution reached the *RecAgr* task, followed by a parallel split gateway. From this gateway, the flow of control is divided into two concurrent execution paths: one path consists of two sequential tasks, *GetIns* and *GenPayIns*, while the other path consists of the *GetTransp* task. This concurrent execution resulted in the sequence of event messages received from the tasks *GetIns*, then *GetTransp*, and finally *GenPayIns*.

The flow of control from both concurrent paths subsequently reaches the exclusive join gateway that allows the flow to proceed only after both execution paths have completed (i.e., in BPMN terminology, when tokens from both incoming sequence flows arrive at the gateway). However, in our implementation of the transformation from a BPMN model to smart contracts, the join gateway is not explicitly implemented. Instead, we channel the incoming sequence flows to the gateway directly to the task that follows the join gateway, the *GenPayTrasnp* task. Of course, the *GenPayTransp* task execution starts only if the control tokens arrive on all incoming sequence flows to it.

As indicated in Figure 3, once the execution of the GenPayTransp task is completed, as expected, the remaining tasks *DoTransp*, *GenRcvePmnt*, and *Success* are executed as confirmed by the event messages shown in the figure.

Execution of the payment services is achieved by invocation of the payment services prepared by the software developer. For each generic payment, the SCaaS transformation from BPMN model causes the invocation of a method of a smart contract prepared by the software developer. The invocation of the payment service is facilitated with the use of a multi-step transaction mechanism supported by SCaaS. The payment service is treated as a multi-step transaction and the generic payment task is just a mechanism for its invocation.

3.6. Discussion

Section 3 demonstrates how the proposed methodology addresses the core objectives established in Section 1, except for reuse and extensibility, which are addressed in Section 4.

Achievement of objective 1—developer preparation of payment services: Section 3.2 established the complete workflow for software developers to prepare payment smart contract services targeting various payment rails. The section demonstrated that developers could model payment services using BPMN fragments, transform them using the SCaaS tool into executable smart contracts, and deploy them to appropriate blockchain environments. The three payment types in the sample use case, on-mainchain crypto-payment (*GenRcvePmnt*), third-party cross-chain payment (*GenPayIns*), and off-chain bank transfer (*GenPayTransp*), illustrate the breadth of payment rails that can be accommodated through this approach.

Achievement of objective 2—business analyst modeling with generic payments: Section 3.5.1 and the accompanying BPMN model in Figure 1 demonstrated that business analysts can represent payment activities using generic BPMN task elements without requiring knowledge of the underlying payment implementation details. The naming convention introduced (task names beginning with "*GenPay*" for payables and "*GenRcve*" for receivables) provides a clear, intuitive mechanism for BAs to indicate payment intent while maintaining technology independence. This separation of concerns enables BAs to focus on business logic and process flow rather than payment implementation complexity, which is the developer's responsibility.

Achievement of objective 3—transformation process with payment service matching: Sections 3.3 and 3.4 detailed the transformation process that bridges generic payment tasks to concrete payment services. The payment service repository mechanism, described in Section 3.3, provides the structured metadata, including semantic descriptors and method signatures, that enables systematic matching based on payment type, currency, blockchain network, and other criteria. Section 3.4 then demonstrated how the SCaaS transformation pipeline augments BPMN models with repository-selected payment fragments and generates smart contracts that invoke pre-deployed payment services rather than regenerating payment logic. Figure 2 illustrated this matching process in practice, showing how the modeler selects the appropriate payment services for each generic payment task during transformation.

In summary, Section 3 established the foundational architecture, methodology, and tooling required to achieve the first three objectives. The sample use case provided concrete evidence that business analysts can model payments generically, developers can prepare reusable payment services, and the SCaaS transformation pipeline can systematically generate smart contracts that orchestrate trade activities while invoking appropriate payment services based on repository-mediated selection.

4. Reuse and Extensibility with Settlement-Netting Payment Services

In this section we focus on reuse and extensibility by creating a payment service that performs settlement netting, also referred to as bulk payments with netting. Reuse allows the same payment service to be used in many trade activities, while extensibility allows creation of a new service while reusing an already existing service without altering

the existing service. We chose the bulk-payment-with-netting service because it involves interaction between parties, interaction that shares the common state of payment execution, and it lends itself for demonstrating the reuse of existing payment services.

We first discuss the settlement-with-netting method, then present and discuss the BPMN model for the trade activity, describe the payment services in terms of its BPMM model and its transformation into a smart contract payment service, and finally we discuss the execution of the blockchain application. We conclude with a brief discission on the achievement of our objectives.

4.1. Settlement with Netting Aka Bulk Payments with Netting

Settlement with netting is used by two trading partners in situations when they have many trade transactions with payments in both directions. To minimize the overhead cost due to services that are charged by the many payments, the parties agree to use settlement netting that aggregates the amount due for each of the parties and nets the cash flows into one payment. In other words, only the net difference in the aggregate amounts is paid or exchanged by the party with the net-owed obligation. Typically, an agreement for the process of settlement netting must be in place, such that:

- All of the payments due between participants are grouped together (bulk).
- The amounts owed between participants are offset (netting).
- The conditions for when netting payments are made are specified.
- Each participant pays or receives one net amount instead of many separate payments.

This approach reduces the number of payments, lowers settlement risk, and improves efficiency in systems like clearing houses, banking, and financial markets [27]. The payment netting agreement must indicate the conditions for when the settlement payments for the net positions are made, such as the settlement is made at the end of each day; when the net position exceeds a certain threshold limit; after a certain number of payments are made; or when some other conditions are met.

As the use case shown in Figure 1 does not lend itself for settlement with netting, we created a simple use case, shown in Figure 4, for two participants, X and Y, making payments to each other, that uses settlement with netting. For simplicity, a participant receives an approved invoice to be paid. After the payment is made, the participant performs a local recording of the result. Although the settlement-with-netting method may be used by two or more parties, for simplicity we use only two participants, X and Y, and each one uses the same BPMN model shown in Figure 4.

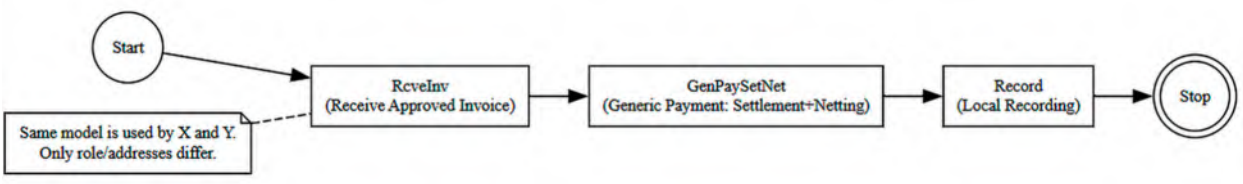


Figure 4. BPMN model for the settlement-with-netting use case.

4.2. Payment Service for Settlements with Netting

The payment service that uses settlement with netting will use an already-existing payment service *StCnPayXY* to facilitate payments between X and Y. We first overview this method and then we proceed with a brief description of the settlement-with-netting method and then describe creation of the payment service that uses settlement with netting.

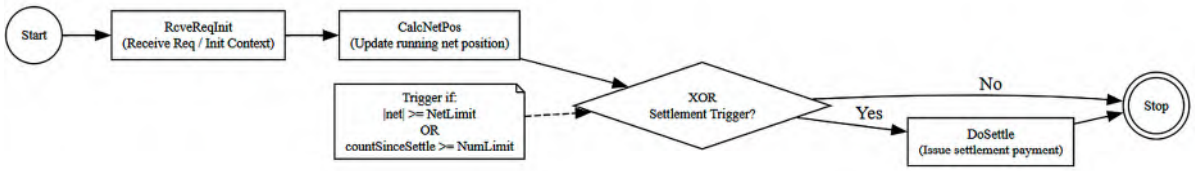
4.2.1. Payments Between X and Y

The *StCnPayXY* payment facilitates the transfer of currency between X and Y. It was created as per the description in Section 3 and then deployed on a 3rd-party stablecoin blockchain, say *StCnBC*, using its cryptocurrency *CrytoCtCn* pegged to some fiat currency. For instance, *StCnBC* might be a USD Coin (Tether) blockchain with its currency USDC (UDTH). The parameters for the *StCnXY* method include account numbers for X and Y, the number of stablecoins to transfer between X and Y, and the direction of payment that identifies which of X and Y is a payee and a payer. Of course, information about the payment service is also stored in the payment repository.

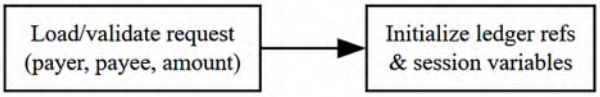
4.2.2. Payment Service That Uses Settlement with Netting

Before creating the payment service, the participants must first agree on the settlement-with-netting details. To simplify exposition, we assume that payments must be made if a net position for a participant exceeds the threshold value *NetLimit*, or if the total number of payments made since the last settlement exceeds the threshold value *NumLimit*.

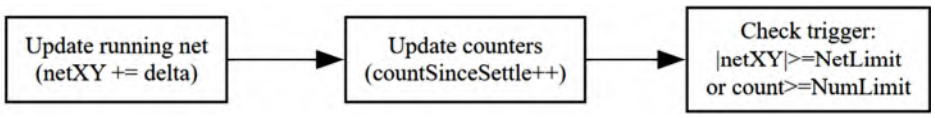
The BPMN model for the payment services is shown in Figure 5, in which part (a) shows the BPMN model, and then parts (b), (c), and (d) show the code for the initialization, calculating the net positions, and making the payment, respectively.



(**a**) BPMN model



(**b**) Receive request initialization *(RcveReqInit)*



(**c**) Netting computation *(CalcNetPos)*

(**d**) Issue settlement payment *(DoSettle)*

Figure 5. BPMN model and implementation for settlement with netting.

The SCaaS tool is used by the software developer to create the payment service, *PaySetNet*, which uses settlement with netting, as was described in Section 3. The BPMN model is created first, which is then transformed using the SCaaS tool into a smart contract. In addition, the payment service information is stored in the repository of payment methods for the *StCnBC* blockchain. The payment service is deployed on the blockchain *StCnBC*, *and* if the payment service is invoked cross-chain; then, the payment service must also have a helper smart contract that provides a bridge for interaction between the *StCnBC* blockchain and the blockchain from which the request for payment originates.

4.3. Trading Activity Smart Contract Generation

In the following, we shall describe how a trading activity that uses the payment service is created and deployed. The BPMN model of the trading activity for X is shown in Figure 4, in which the generic payment *GenPaySetNet* needs to be realized using the payment service *PaySetNet*, which is deployed on the blockchain *StCnBC*. Before we can instantiate the *GenPaySetNet* service, if the information about the *PaySetNet* service is not in the repository of payment services used by X, it must be imported, and it must be ensured that a helper smart contract exists on the *XpBP* chain to collaborate with the helper smart contract on the payment chain *StCnPayBC* in order to provide a bridge for interaction between the two chains.

Once the payment repository for *XpBC* contains information on the payment service *PaySetNet*, generation of the smart contract and the APIs using the SCaaS tool proceeds as described in Section 3. During the transformation, the modeler selects the *PaySetNet* payment service from the repository as instantiation for the generic *GenPaySetNet* payment, and the transformation produces the smart contract and API. Figure 6 shows that the developer chose the *PaySetNet* payment service to be used to instantiate the *GenPaySetNet* generic payment service.

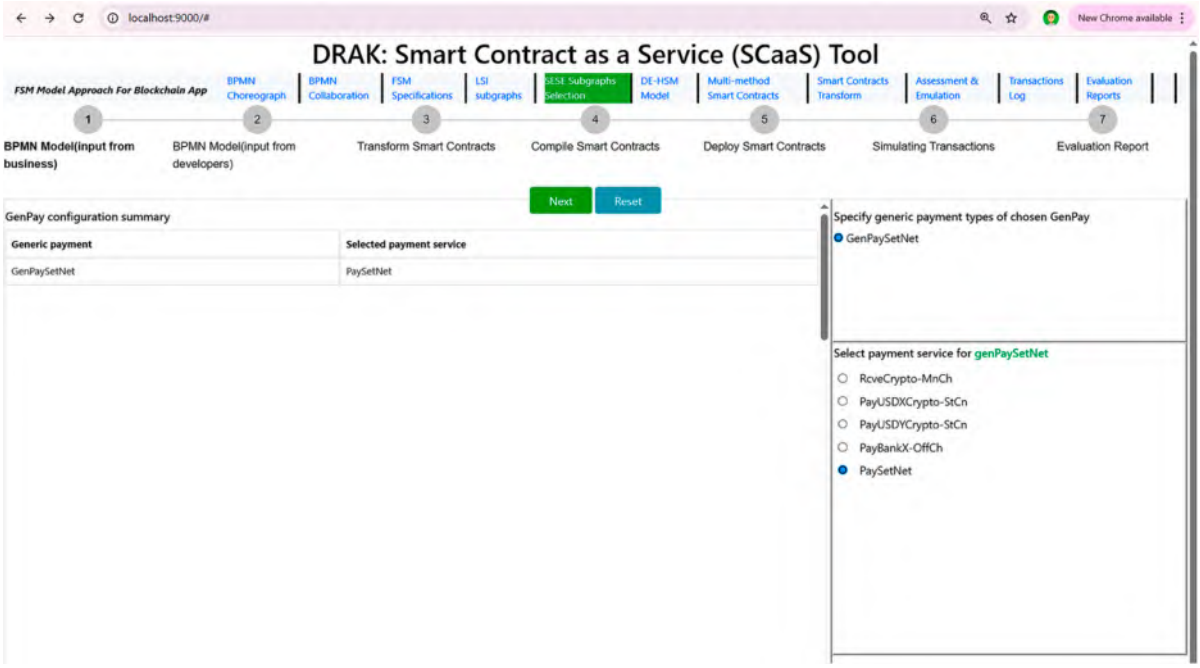


Figure 6. Selection of the actual payment service to be used for the *GenPaySetNet*.

Generation of the smart contract and the API by partner Y for the model shown in Figure 4 follows the same process as for the partner X.

4.4. Execution

Recall that we described the execution environment in Section 3.5.2. We created smart contracts for partners X and Y, as well as a smart contract implementing the payment service. The smart contracts for partner X and partner Y were deployed on blockchains *XpBC* and *YpBC*, respectively, where *XpBC* is an Ethereum-based blockchain, and *YpBC* is a Hyperledger Fabric (HLF)-based blockchain.

The payment smart contract *PaySetNet*, which is responsible for maintaining the net position and determining when settlement should be triggered, was deployed on the Ethereum-based payment service chain, *StCnBC*.

Also recall that the payment service *StCnPayXY*, which is used to perform the actual transfer of stablecoins between partners X and Y upon settlement, is deployed on the same payment service chain, *StCnBC*, and is invoked via the payment helper bridge smart contracts.

Furthermore, for simplicity, we chose *NetNum* = 2, so we could observe the payments being made. We chose *NetLimit* arbitrarily to be 100. We chose step-by-step execution for testing and used different values for *NetNum* and *NetLimit* to ensure that netting is performed correctly and payments are issued appropriately.

Figure 7 shows an architectural diagram of the software components involved when the payment service is invoked by the trading activity smart contracts for X and Y. As indicated before, we set the threshold value *NetNum* = 2 and chose bulk testing by issuing multiple transactions. We should observe that after two payments are invoked, a payment should be performed by transfer of funds between X and Y to achieve a zero net position. Figure 8 shows a window containing the event messages generated while executing the blockchain application involving smart contracts for the trade activities of X and Y and payment services.

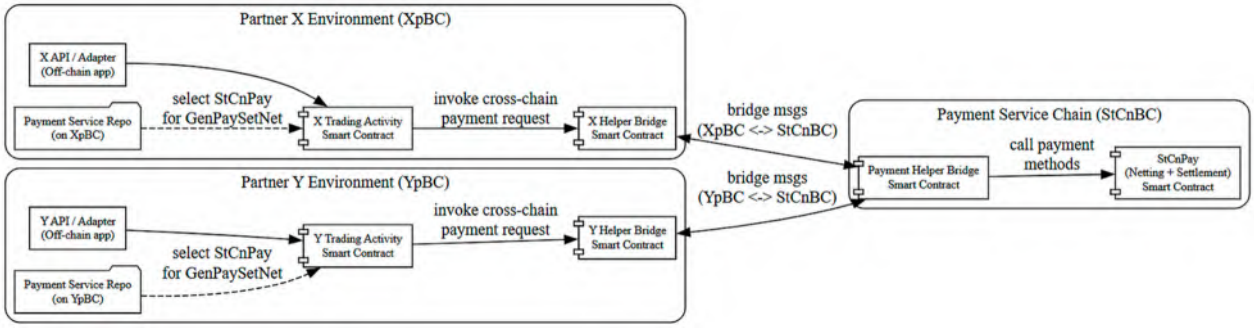


Figure 7. Architectural diagram of the software components' interaction to realize the payment service.

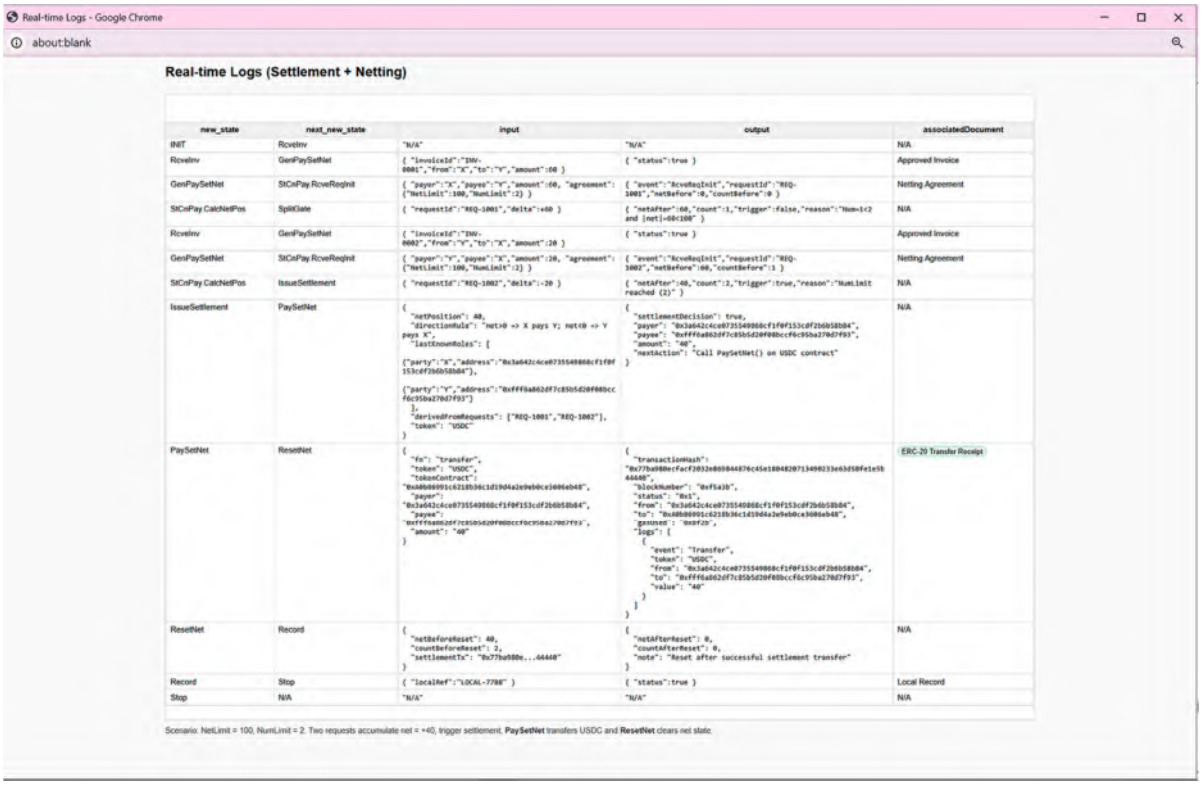


Figure 8. Execution trace using event messages emitted by smart contracts.

After initialization represented by the INIT event, an event message is received from the execution of the *RcveInv* task by the partner X, which is followed by the execution of the *GenPaySetNet* task by X. As a result, the payment service, for X to pay Y, is invoked, which is followed by interaction between the helper smart contracts, wherein the helper smart contract deployed on the payment chain, *StCnBC*, validates the request and performs initialization of the payment service. As this is performed by the helper smart contract, there is no event message that is emitted that corresponds to the *RcveReqInit* task shown in Figure 8. Thus, the next event message received is from the task *StCnPay.CalckNetPos*. As settlement payment is not required, execution of the payment service completes.

The next two event message are emitted by the trade activity smart contract for Y, and they are similar to those of X: an event message is received from the execution of the task *RcveInv* by the partner Y, which is followed by an event message from the task *GenPaySetNet* to request a payment to be made by Y. The helper smart contracts on the *YpBC* and *StCnBC* chains interact, with the helper smart contract on the chain *StCnBC* performing the request validation, initialization, and then causing the execution of the task *StCnPay.CalcNetPos*. As this is the second payment request and *NetNum* = 2, after calculating the net position, the task *IssueSettlement* is executed, which invokes the payment service *StCnPayXY*. This is then followed by a task to reset the net position. Finally, the last task of the payment service, *Record*, is executed.

4.5. Discussion

Section 4 specifically addresses the reuse and extensibility objectives identified in Section 1, demonstrating how payment services can be shared across multiple trade processes and how new payment services can be constructed by composing existing services without modification.

Achievement of reuse objective: The settlement with netting payment service (*PaySetNet*) exemplifies reuse in two distinct dimensions. First, it demonstrates service-level reuse by invoking the pre-existing *StCnPayXY* payment service for the actual fund transfers between parties X and Y. Rather than reimplementing token transfer logic, *PaySetNet* delegates payment execution to *StCnPayXY*, treating it as a primitive operation. Second, the section showed process-level reuse: both trading partners X and Y use identical BPMN models (Figure 4) and select the same *PaySetNet* payment service from the repository (Figure 6). This shared usage across multiple trading activities illustrates how payment services once developed and deployed become reusable assets available to any trade process requiring similar payment functionality.

Achievement of extensibility objective: The *PaySetNet* service demonstrates extensibility by composing sophisticated payment logic—including net position tracking, threshold-based settlement triggering, and conditional payment execution—while building upon the simpler *StCnPayXY* service. Critically, this composition occurred without any modification to the existing *StCnPayXY* service or its repository entry. The developer created *PaySetNet* by modeling new BPMN logic (Figure 5) that incorporates initialization, netting computation, and settlement decision logic, then invokes *StCnPayXY* when settlement is required. This compositional approach enables incremental addition of payment capabilities: as new payment patterns emerge (e.g., escrow, conditional release, multi-signature authorization), developers can implement them as new services that orchestrate existing payment primitives, thereby extending the payment service ecosystem without disrupting deployed services.

5. Related Work

We first provide the literature related to our approach for the generation of smart contracts from BPMN models, and then we provide the literature related to the work in this paper on blockchain-based payment processing.

5.1. Generation of Smart Contracts from BPMN Models

Several research initiatives have explored transforming Business Process Model and Notation (BPMN) models into executable smart contracts. The Lorikeet project employs a two-phase transformation approach [28]. In the initial phase, the BPMN model undergoes analysis and transformation into smart contract methods, which are subsequently deployed on the Ethereum blockchain. An off-chain component manages communication between the decentralized application (Dapp) and external actors. Participants exchange messages according to the BPMN choreography, with an off-chain component coordinating these exchanges. The smart contract incorporates a monitoring mechanism that records and enforces the choreography model, ensuring message exchanges follow the predetermined sequence. Additionally, Lorikeet supports asset management capabilities, including both fungible and non-fungible tokens, providing a registry and methods for asset operations such as transfers. This functionality enables rapid prototyping of smart contracts from BPMN models requiring asset management features, facilitating iterative development, testing, and refinement before final deployment.

The Caterpillar framework presents an alternative methodology, concentrating on BPMN models within a single pool, which is a BPMN structural construct, where all business processes are recorded on the blockchain [18]. Its architecture comprises three distinct layers: the web portal, the off-chain runtime, and the on-chain runtime. The on-chain runtime layer encompasses smart contracts responsible for workflow control, interaction management, configuration, and process oversight, with Ethereum serving as the primary blockchain platform.

Loukil et al. introduced the collaborative business process (CoBuP) architecture for executing collaborative business processes on blockchain [29]. Unlike approaches that directly compile BPMN models into smart contracts, CoBuP deploys a generic smart contract that invokes predefined functions. The architecture features three layers: conceptual, data, and flow. BPMN models are transformed into a JSON workflow representation that governs process instance execution and manages interactions with blockchain data structures.

Di Ciccio, Cecconi and Mendling conducted a comparative analysis of the Lorikeet and Caterpillar approaches across multiple dimensions, including model execution capabilities, BPMN element coverage, detection of incorrect behaviors, sequence enforcement mechanisms, participant selection, access control, and asset management [30]. Their analysis highlighted the distinctive characteristics of each approach and established a foundation for systematic comparison.

Possik et al. addressed the issue of developing a methodology for the Distributed Simulation of High-Level Architecture (HLA) of interacting systems of which processes are represented using BPMN models [31]. They show that BPMN-based systems cannot be treated as monolithic simulations, but that message exchange, event ordering, and time synchronization become first-class concerns that must be modeled using discrete-event mechanisms and emphasize modular DES components coordinated through a federation infrastructure. They thus provide the conceptual motivation for treating BPMN interactions as discrete-event coordination problems which we adopt into the BPMN-to-smart contract transformation pipeline.

We use the above insights to utilize the DE part of the DE-HSM model to explicitly represent BPMN message flows and synchronization semantics between processes, rather

than embedding them implicitly in control flow. Additionally, we separate interaction logic (DE layer) and functional behavior (HSM layer), enabling independent evolution and reuse realized as modular smart contracts and payment services.

This transformation enables systematic analysis of process flows to identify localized processing using single-entry single-exit (SESE) subgraphs. In contrast to systems that directly translate BPMN models into smart contract code, our approach generates smart contracts as abstract representations of process flows, with implementation details expressed through concurrent FSMs. The process logic executes through a smart contract that manages FSM state transitions, with specific transitions triggering the execution of localized BPMN tasks. Each task is implemented as a smart contract method with clearly defined inputs and outputs. Furthermore, localized SESE subgraphs can be packaged and deployed as independent smart contracts, potentially on sidechains for improved efficiency, or utilized to define nested trade transactions with formally specified transactional properties.

To the best of our knowledge, no prior formal work has addressed multi-step, multi-method transactions for blockchain smart contracts, what we term trade transactions. These multi-step transactions contain executions of multiple independently invoked smart contract methods by different actors.

5.2. Smart Contract Upgradeability and Repair

The immutability property of blockchain technology, while promoting trust, creates challenges when collaborative activities require modifications to repair security issues or incorporate new features. Smart contract upgradeability has attracted substantial research attention, as evidenced by comprehensive literature surveys on the topic [32–34]. There are several bug-fixing frameworks, e.g., EVMPatch [35], SolSaviour [36], FlexChain [37,38], and BlockME [39].

The work most closely aligned with our repair approach is that of Klinger et al. [40], which analyzes and implements three upgradeability concepts: a registry-based approach, a proxy pattern, and a combined registry with pattern segregation [40]. Their use case involves a large organization with multiple departments, where the BPMN model of collaboration transforms into smart contracts, with each department's activities represented by a separate contract. Upgrade management must ensure consistency between previously executed activities and the upgrade context. Their findings indicate that the Unstructured Storage Proxy pattern shows the most promise for practical deployment, particularly regarding cost-effectiveness and minimal added complexity.

Compared to Klinger et al., the TABS+R approach naturally packages activities into separate smart contracts, exploiting nested trade transactions to protect against inconsistencies arising when some activities complete using the old contract version while others execute with the newly repaired or upgraded contract [40].

5.3. Payment Rails for Smart Contracts in Blockchain Applications

Our work on payment rails for smart contracts builds upon the above research trajectory and intersects more specifically with the literature on blockchain-based payment processing, settlement mechanisms, and financial workflows modeled as business processes.

A substantial body of research explores blockchain-enabled payment systems and settlement infrastructures, often focusing on performance, trust minimization, or cross-organizational coordination. However, many such systems are designed directly at the protocol or smart contract code level, without leveraging high-level business process models. As a result, the alignment between business requirements and deployed contract logic remains difficult to verify or evolve.

Within the BPMN-to-smart contract literature, most prior work treats payments as simple task executions or token transfers embedded within a broader workflow [41,42]. These approaches typically lack explicit modeling of payment-specific concerns such as settlement modes, batching, netting, or conditional execution based on financial thresholds.

This approach to blockchain-enabled payments extends existing BPMN-driven smart contract generation by explicitly focusing on payment orchestration and settlement logic as first-class process constructs. In doing so, it complements our earlier SCaaS/TABS-based work by refining how payment-related BPMN patterns are identified, transformed, and executed on blockchain platforms. This focus aligns with emerging research on blockchain-based financial infrastructure, while retaining the model-driven advantages of our approach to the transformation of BPMN models to smart contracts.

Furthermore, our work distinguishes itself from generic payment-contract frameworks by integrating payment logic with transactional guarantees and lifecycle management mechanisms inherited from TABS+ and TABS+R. This integration allows payment processes to be executed, monitored, repaired, or upgraded in a manner consistent with the original BPMN specification, addressing limitations of static or hard-coded payment smart contracts.

In this sense, our work on payment rails occupies a niche at the intersection of BPMN-based smart contract generation and blockchain payment systems.

6. Summary, Contributions, Future Work and Conclusions

6.1. Summary

This paper addressed the problem of systematically integrating payment rails into smart contract-as-a-service (SCaaS) solutions generated from BPMN models. While prior work by the authors established a model-driven approach for transforming BPMN business processes into executable smart contracts, payment handling in those solutions remained largely implicit or implementation-specific. Given the central role of payments in inter-organizational processes, supply chains, and financial workflows, the lack of explicit modeling and transformation support for payment mechanisms represents a significant gap.

The primary objective of this research was to define, model, and operationalize payment rails within BPMN-to-smart contract transformations. To achieve this objective, the paper analyzed common enterprise and financial payment patterns, including real-time gross settlement, deferred net settlement, and bulk or threshold-based settlement, and examined how these patterns can be formally represented at the BPMN level and correctly realized at the smart contract level.

Building on the SCaaS architecture and transformation pipeline developed in earlier work, the paper proposed extensions to BPMN modeling conventions, execution semantics, and smart contract generation logic to explicitly capture payment behavior. The resulting approach ensures that the payment execution, settlement, reconciliation, and netting semantics are preserved across the model-to-code boundary, while remaining adaptable to different blockchain platforms and off-chain payment infrastructures.

6.2. Contributions

The contributions of this paper are:

- *Separation of concerns between trade activity modeling and payment implementation:* Sections 3.2 and 3.5 established a clear architectural separation where business analysts model trade activities and generic payments in BPMN, while software developers independently prepare reusable payment service smart contracts. This division of responsibilities enables domain experts to focus on business logic without requiring deep knowledge of payment rail implementation details.

- *Repository-mediated transformation:* Section 3.3 introduced the payment service repository as a critical intermediary that maintains structured metadata about deployed payment services, including semantic descriptors and method signatures. This repository enables automated matching and selection during transformation, serving as the knowledge base that connects business-level payment intent with technical payment execution.
- *Multi-rail payment support:* The sample use case in Section 3 demonstrated heterogeneous payment scenarios spanning on-chain native payments, third-party cross-chain stablecoin payments, and off-chain conventional banking transfers. This diversity illustrates the methodology's adaptability to different business needs and technical constraints.
- *Reuse and composition of patterns for extensibility of payment services*: Section 4 established that complex payment services can be constructed by orchestrating simpler payment primitives through BPMN modeling and SCaaS transformation. The *PaySetNet* service demonstrated this by building sophisticated netting logic while invoking the existing *StCnPayXY* service, showing a clear path for extensibility without code duplication or the modification of existing services.
- *Shared state management across participants:* The *PaySetNet* service, in Section 4, illustrated how payment services can maintain a shared state (net positions for X and Y) that persists across multiple invocations from different trade processes and different participants, enabling sophisticated multi-party settlement patterns.
- *Cross-platform payment service invocation:* Section 4's use case validated that payment services deployed on one blockchain can be reliably invoked from trade processes executing on heterogeneous blockchain platforms (Ethereum-based and Hyperledger Fabric-based), with the repository and helper contract infrastructure transparently managing platform-specific communication details.
- *Incremental ecosystem growth:* By showing how *PaySetNet* was added to the repository after development and immediately became available for use by both X and Y's trade processes, Section 4 demonstrated the practical mechanics of payment service ecosystem expansion—new capabilities become available to all participants without requiring updates to existing services or trade processes.

6.3. Future Work

Although our approach to simplifying the development of smart contracts for the trade of goods and services has made meaningful progress, several challenges and limitations remain. In this subsection, we outline these limitations and describe our plans to address them.

6.3.1. Identity Management for Smart Contracts in the Trade of Goods and Services

Identity management is a critical component of smart contract-based systems for the trading of goods and services as it enables reliable identification, authentication, and authorization of participating entities. In blockchain environments, identities are typically represented by cryptographic key pairs, allowing parties to authenticate and authorize transactions without relying on centralized intermediaries. While this model supports decentralization and pseudonymity, it is often insufficient for commercial trade scenarios that require legal accountability or regulatory compliance.

To address these requirements, smart contract systems frequently distinguish between identity authentication and attribute verification. Authentication confirms control over a cryptographic identifier, whereas attribute verification establishes whether an entity satisfies specific properties relevant to the transaction, such as organizational status, licensing, geographic location, or compliance with KYC/AML requirements [7,43]. Smart contracts

typically depend on trusted third parties or oracles, such as identity providers, certification authorities, or regulators, to attest to these attributes and make them available for on-chain enforcement.

Identity management also underpins access control and compliance enforcement in smart contracts. Contracts may restrict participation to authorized identities, enforce role-based permissions, or condition execution and payment on verified identity attributes. This is particularly important for external compliance, where regulations mandate that only eligible or verified entities engage in certain trade activities [44]. At the same time, internal compliance relies on identity mechanisms to ensure that contractual actions are executed by the correct parties in accordance with agreed roles.

Several identity frameworks and service providers support these requirements in practice. Prominent examples include Sovrin and Hyperledger Indy/Aries, which focus on decentralized identity and verifiable credentials; uPort/Veramo (ConsenSys) and Civic, which provide blockchain-based identity verification services; and enterprise-oriented solutions such as Microsoft Entra Verified ID and Trinsic, which integrate decentralized identity concepts with existing organizational identity infrastructures [45]. We are investigating how to incorporate identity management into our transformation pipeline.

6.3.2. Compliance

In the context of smart contracts for the trade of goods and services, payment services and compliance enforcement constitute two closely related, yet analytically distinct, functions. Smart contracts facilitate automated payment services by executing transfers of value once predefined contractual conditions are met, such as delivery confirmation or the completion of contractual milestones. These capabilities enhance transactional efficiency and transparency, particularly in cross-border and multi-party commercial settings.

Compliance enforcement, by contrast, focuses on ensuring that contractual performance complies with applicable legal, regulatory, and policy requirements. Smart contracts can support a priori compliance by embedding rule-based conditions that must be satisfied before payments, or other contractual actions, are executed. This enforcement encompasses both the internal and external dimensions of compliance. Internal compliance refers to adherence to the contractual obligations agreed upon by the parties, such as performance criteria, delivery schedules, or usage constraints [7]. External compliance concerns obligations imposed by law or regulation, including know-your-customer (KYC) and anti-money laundering (AML) requirements, sanctions regimes, tax obligations, and sector-specific licensing requirements [6,44]. By encoding such constraints, smart contracts can automatically prevent, delay, or adjust transactions that would otherwise result in non-compliant outcomes, while simultaneously generating immutable audit trails.

Although payment services and compliance enforcement are closely intertwined in practice, they should be treated separately at the conceptual and design levels. Payment services are primarily concerned with the efficient and reliable transfer of value, whereas compliance enforcement emphasizes legality, risk mitigation, and accountability. Preserving this distinction supports modular system design, allowing compliance rules, particularly those driven by external regulatory changes, to be updated or adapted without disrupting core payment functionality. At the same time, effective operation requires close integration: compliance conditions must directly govern payment execution to achieve meaningful enforcement.

We are currently addressing the design challenges associated with enforcing compliance prior to payment execution. In particular, we are investigating how to effectively and efficiently determine which internal and/or external compliance rules apply to a given trade activity, and how to verify that all applicable requirements are satisfied before payment is authorized.

6.3.3. Securing Smart Contract Methods

To secure smart contracts, we adopt the approach proposed by Mavridou et al. [1,46], which hardens smart contracts generated through the transformation of finite state machines (FSMs) into smart contract methods. Following this transformation, each method is secured by inserting established security patterns that protect against (i) reentrancy attacks through locking mechanisms, (ii) transaction-ordering vulnerabilities arising from unpredictable states, (iii) unsafe timed transitions, and (iv) unauthorized access.

We have successfully incorporated reentrancy protection into our tool by inserting appropriate locking patterns into the generated smart contracts and by supporting access control mechanisms. In practice, these security patterns are injected at the beginning and end of each smart contract method. In future work, we plan to develop additional smart contract patterns to guard against all known classes of smart contract vulnerabilities. Furthermore, unlike native blockchain transactions, trade transactions must also be protected against man-in-the-middle attacks, which introduce additional security considerations.

6.3.4. Validation and Verification

Validation and verification are integral components of the transformation process from BPMN models to deployed smart contracts. The transformation of BPMN models produces a DE-FSM model, which can be interpreted as a transition system. We plan to apply the VerySolid verification framework [47] to ensure that the generated smart contracts are correct by design.

Currently, BPMN modelers are required to document the information that flows along the execution paths of the model. We plan to extend this documentation to include the desired system properties, which will then be formally verified. Specifically, we aim to verify properties such as liveness, reachability, deadlock freedom, and the satisfaction of specified correctness requirements, including those related to tasks executed off-chain.

6.3.5. Discussion

Future work will focus on transitioning from technical validation to evaluating the adoption and practical usability of the proposed approach. However, such evaluation can only be meaningfully undertaken after the technical challenges identified above, particularly those related to compliance enforcement, security, identity management, and verification, have been sufficiently addressed. Once these foundational issues are resolved, attention can shift toward assessing adoption by end users, business analysts, and developers as a prerequisite to commercialization. Evaluating adoption poses its own challenges, as it requires attracting developers willing to invest time and effort to experiment with and provide feedback on a new tool. Without direct incentives or compensation, encouraging sustained participation from the developer community is inherently difficult. Accordingly, future work will need to explore appropriate strategies and incentives to support adoption studies and inform the path toward commercialization.

6.4. Conclusions

This research demonstrates that payment handling in smart contract-based systems cannot be treated as a secondary or purely technical concern. Instead, payment rails must be explicitly modeled and systematically transformed alongside core business logic to ensure correctness, compliance, and interoperability. By elevating payment rails to first-class citizens in BPMN-driven SCaaS solutions, the proposed approach improves transparency, reduces ambiguity between business intent and technical implementation, and supports a broader range of real-world enterprise use cases.

The results show that a model-driven, payment-aware SCaaS architecture can accommodate diverse settlement mechanisms without sacrificing automation or scalability. Moreover, the separation of concerns between business process modeling, payment semantics, and execution infrastructure enables greater flexibility in adapting solutions to evolving regulatory, technological, and organizational requirements.

Author Contributions: Conceptualization, C.G.L., P.B. and D.J.; Methodology, C.G.L., P.B. and D.J.; Software, C.G.L. and P.B.; Validation, C.G.L. and P.B.; Formal analysis, C.G.L. and P.B.; Investigation, C.G.L.; Writing—original draft, C.G.L., P.B. and D.J.; Writing—review and editing, C.G.L., P.B. and D.J.; Project administration, D.J. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: Data used to generate the figures, such as the data models and event messages generated by software execution, are available in data files stored at https://web.cs.dal.ca/~chris/future_internet/ (accessed on 18 January 2026).

Conflicts of Interest: The authors declare no conflict of interests.

Appendix A

The appendix shows information that supplements the main text with some details:

- Appendix A.1. provides further information on the helper smart contracts described in Section 3.5.3.
- Appendix A.2. provides supplementary information and screenshots related to how the SCaaS tool is used.

Appendix A.1. Helper Smart Contracts

When the mainchain smart contract invokes a payment service deployed cross-chain or off-chain, it requires the use of a helper smart contract or an oracle to provide a bridge to the crypto-payment blockchain: the helper smart contract method facilitates the interaction between the main smart contract and the target payment service, which is deployed on a different blockchain or implemented as an off-chain service, requesting it to make the payment and then receiving from the payment service indication of the success/failure that is communicated to the main smart contract.

Upon receiving the reply from the payment blockchain, the helper smart contract informs the trade activity whether the payment succeeded or failed. The BPMN fragment model is shown in Figure A1. The trade activity smart contract invokes the helper smart contract method that provides a bridge to support invocation of the helper smart contract on the crypto-payment blockchain.

Once the payment is performed, the helper smart contract on the payment bridge interacts with the helper smart contract on the blockchain with the trade activity smart contract. Thus, in Figure A1, the helper smart contract is shown to be included in a different swimlane, which is a BPMN term, represented in the figure as a rectangle encompassing the BPMN model elements representing the execution of the helper smart contract. The dashed arrow labeled "Perform Payment/Bridge Steps" is used to represent payment activities executed on either the payment blockchain or off-chain. Completion of such activities concludes with communicating the success/failure to the helper smart contract executing on the blockchain with the trade activity smart contract. Recall that for the cross-chain payment service, the developer must ensure that there is a helper smart contract on both blockchains involved, the trade activity and payment blockchains.

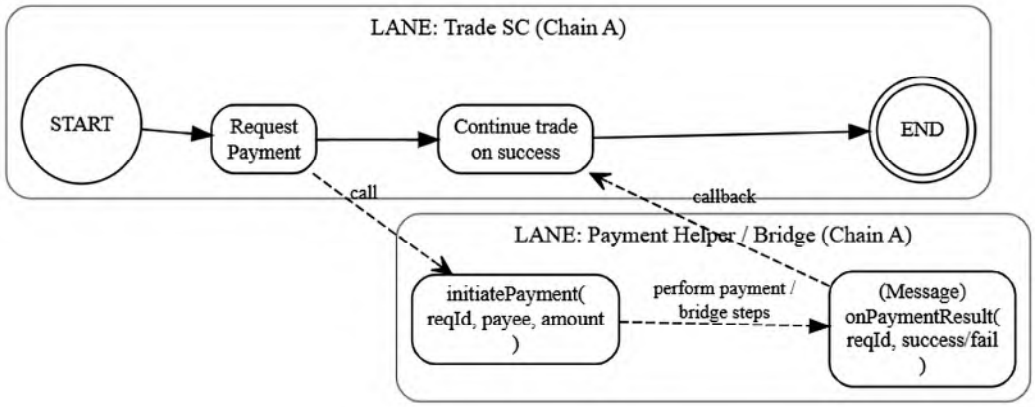


Figure A1. Request–reply interaction with the helper smart contract for cross-chain payment.

Appendix A.2. SCaaS Tool: Model Creation, Transformation, and Execution

The SCaaS tool utilizes the Camunda platform (Camunda, 2025) for creating BPMN models in accordance with the BPMN specifications (OMG 2013, 2023) and stores the created BPMN models in XML format. Figure A2 shows a screenshot of the SCaaS tool during BPMN model creation for the example application. Once a BPMN model is saved in XML form, it is transformed through a sequence of steps controlled via tabs in the tool's user interface at the top of the screen. The initial tabs support BPMN modeling, while subsequent tabs guide the user through the transformation process, including smart contract generation, deployment, and execution control.

Assuming that the required payment smart contract services have been prepared by the software developer, the BPMN model shown in Figure A2 has been created, and the user initiates the transformation, the BPMN model is first converted into a directed-graph representation of a DE-HSM model. This model is analyzed to identify fragments suitable for treatment as multi-step transactions. Next, for each generic payment task, the tool prompts the modeler to select which of the available deployed payment services should be used to realize that task, as described in Section 3.5.3.

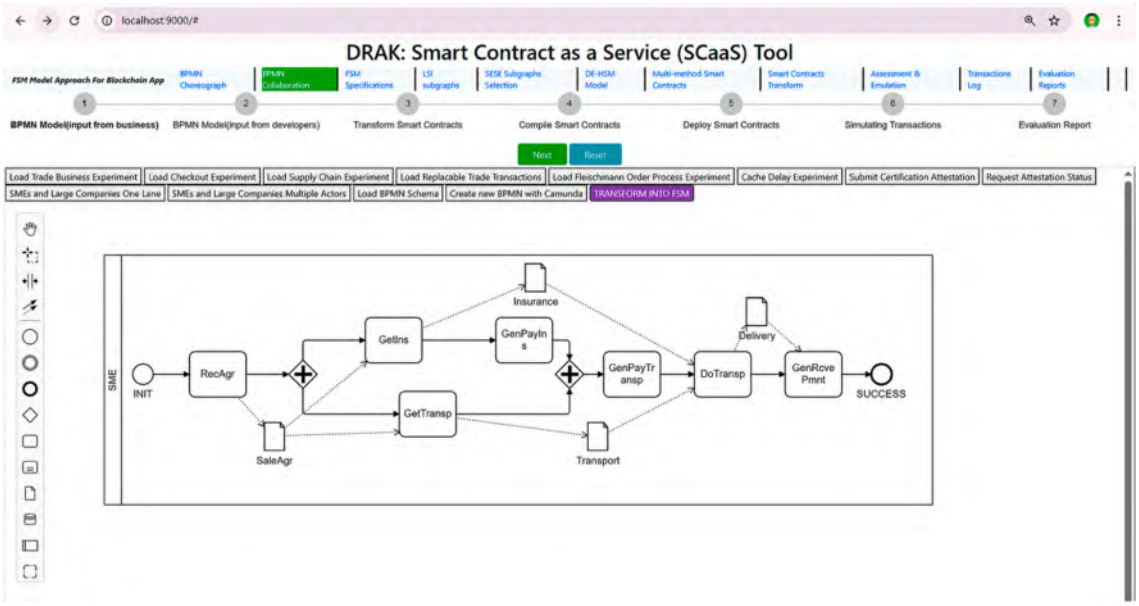


Figure A2. Screenshot of the SCaaS tool preparing the BPMN model for the sample use case.

Once the transformation of the BPMN model results in the deployment of the smart contract for the trade activity and the APIs are instantiated, execution is controlled by the

SCaaS tool that has been built for research experimentation and evaluation purposes with the objective of achieving proof-of-concept. The tool thus provides functionalities targeted to the development of software and its evaluation rather than execution in a production environment. The modeler can examine the generated system execution by stepping through the execution message-by-message, while the tool shows the progress graphically by showing the change to states of the individual FSMs representing the DE-FSM sub-models. This feature is helpful in testing and manual verification. Delays are also shown as execution proceeds step-by-step. The tool also provides functionality for bulk issuance of transactions, so that the modeler can instruct the tool to measure average execution delays when trading activities are issued with a selected frequency so that execution-delay properties may be measured.

Figure A3 shows a screenshot of the tool at the beginning of the execution. At the top of the screen, underneath the title, are the horizontal tabs that control the different phases and sub-phases of the transformation. Below the tabs, there are centrally located buttons for controlling the execution, namely Next and Reset, and on a line below and to the left are the Start Evaluation and Stop Evaluation buttons. To the left and slightly below the Start Evaluation button, there are selections for either: (i) Load testing, when many trade activities are executed to derive averages such as average delays, or (ii) Take Steps for stepping through the execution to view details when testing the software using manual verification.

Below the Start Evaluation and Stop Evaluation buttons and the Load-testing or Step-through options, on the left-hand side, are the main components of the DE-HSM model shown as rectangles, namely, the smart contracts for the main trading activity, and below, the three smart contract payment methods. In the middle, we show the selected payment service, wrapped by the corresponding generic payment service that is identified by the BPMN model. Below, information about communication with the IPFS is shown, as an IPFS is used to store all of the documents involved in trade activity processing. Below this, information is shown about the invocation of HTTP services, such as communication with external APIs or with an IPFS for storage services.

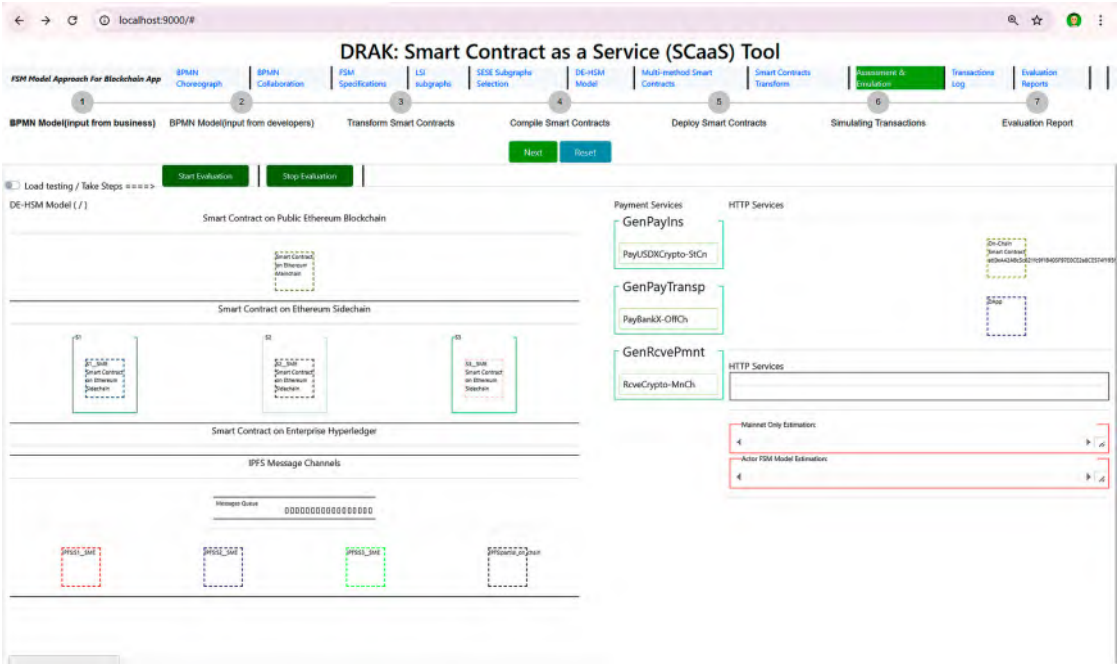


Figure A3. SCaaS tool—screenshot of SCaaS for controlling and observing execution.

References

1. Mavridou, A.; Laszka, A. Designing secure Ethereum smart contracts: A finite state machine based approach. In *Financial Cryptography and Data Security (FC)*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2018; Volume 10957, pp. 523–540.
2. Liu, C.G.; Bodorik, P.; Jutla, D. Automated mechanism to support trade transactions in smart contracts with upgrade and repair. *Blockchain: Res. Appl.* **2025**, *6*, 100285. [CrossRef]
3. Bank for International Settlements. *Principles for Financial Market Infrastructures*; BIS: Basel, Switzerland, 2012.
4. SWIFT. Standards MT: General Information. 2020. Available online: https://www2.swift.com/knowledgecentre/rest/v1/publications/usgi_20200724/4.0/usgi_20200724.pdf (accessed on 12 March 2025).
5. Narayanan, A.; Bonneau, J.; Felten, E.W.; Miller, A.; Goldfeder, S. *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction*; Princeton University Press: Princeton, NJ, USA, 2016.
6. Buterin, V. A Next-Generation Smart Contract and Decentralized Application Platform. *Ethereum White Paper*. 2014. Available online: https://ethereum.org/en/whitepaper/ (accessed on 4 February 2025).
7. Szabo, N. Formalizing and securing relationships on public networks. *First Monday* **1997**, *2*, 9. [CrossRef]
8. Narayanam, K.; Ramakrishna, V.; Nishad, S. Atomic cross-chain exchanges of shared assets. In *Proceedings of the 4th ACM Conference on Advances in Financial Technologies (AFT)*; Association for Computing Machinery: New York, NY, USA, 2022; pp. 148–160.
9. Zamyatin, A.; Al-Bassam, M.; Zindros, D.; Kokoris-Kogias, E.; Moreno-Sanchez, P.; Kiayias, A.; Knottenbelt, W.J. SoK: Communication across distributed ledgers. In *Financial Cryptography and Data Security (FC)*; Springer: Berlin/Heidelberg, Germany, 2021.
10. Wood, G. Polkadot: Vision for a Heterogeneous Multi-Chain Framework. *Web3 Foundation*. 2016. Available online: https://polkadot.network/whitepaper/ (accessed on 18 January 2026).
11. Garratt, R.; Shin, H.S. Stablecoins: Risks, potential and regulation. In *BIS Quarterly Review*; BIS: Basel, Switzerland, 2020.
12. Belchior, R.; Vasconcelos, A.; Guerreiro, S.; Correia, M. A survey on blockchain interoperability: Past, present, and future trends. *ACM Comput. Surv.* **2021**, *54*, 168. [CrossRef]
13. Pahl, C.; El Ioini, N.; Helmer, S. Semantic richness in financial transactions. *IEEE Softw.* **2019**, *36*, 16–22.
14. Weber, I.; Staples, M. Programmable Money: Next-generation Conditional Payments using Blockchain. In Proceedings of the 11th International Conference on Cloud Computing and Services Science (CLOSER 2021), Prague, Czech Republic, 23–25 April 2021; pp. 7–14.
15. *ISO/IEC Std. 19510:2013*; Information Technology—Object Management Group Business Process Model and Notation (BPMN). International Organization for Standardization and International Electrotechnical Commission: Geneva, Switzerland, 2013.
16. Object Management Group. *Decision Model and Notation (DMN), Version 1.5*; OMG Standard: Milford, MA, USA, 2023.
17. López-Pintado, M.; García-Bañuelos, L.; Mendling, J.; Weber, I. Caterpillar: A business process execution engine on the Ethereum blockchain. In *Business Process Management (BPM)*; Springer: Cham, Switzerland, 2017; pp. 87–103.
18. López-Pintado, M.; García-Bañuelos, L.; Weber, I. Lorikeet: A model-driven engineering tool for blockchain-based business processes. *Softw. Pract. Exp.* **2019**, *49*, 1453–1477. [CrossRef]
19. Weber, I.; Xu, X.; Riveret, R.; Governatori, G.; Ponomarev, A.; Mendling, J. Untrusted business process monitoring and execution using blockchain. In *Business Process Management (BPM)*; Springer: Cham, Switzerland, 2016; pp. 329–347.
20. Dalhousie Blockchain Lab. Dalhousie Blockchain Lab: Executive Summary. 2024. Available online: https://blockchain.cs.dal.ca (accessed on 15 October 2025).
21. Gudgeon, L.; Moreno-Sanchez, P.; Roos, S.; McCorry, P.; Gervais, A. SoK: Off the Chain Transactions. IACR Cryptology ePrint Archive, Paper 2020/112, 2020. Available online: https://eprint.iacr.org/2019/360.pdf (accessed on 15 October 2025).
22. W3C. Semantic Annotations for WSDL and XML Schema (SAWSDL). W3C Recommendation REC-sawsdl-20070828, 2007. Available online: https://www.w3.org/TR/sawsdl/ (accessed on 15 October 2025).
23. Liu, C.G.; Bodorik, P.; Jutla, D. Smart contracts for SMEs and large companies. In *Proceedings of the 2024 IEEE Virtual Conference on Communications (VCC)*; IEEE: Piscataway, NJ, USA, 2024; pp. 1–7. [CrossRef]
24. Liu, C.G.; Bodorik, P.; Jutla, D. BPMN model to smart contract by business analyst. In *Proceedings of the 2025 5th Intelligent Cybersecurity Conference (ICSC)*; IEEE: Piscataway, NJ, USA, 2025; pp. 122–129. [CrossRef]
25. DigitalOcean. Available online: https://www.digitalocean.com/ (accessed on 18 January 2026).
26. Camunda. Camunda Platform: Workflow and Decision Automation. Available online: https://camunda.com/ (accessed on 18 January 2026).
27. Kenton, W. Multilateral Netting. *Investopedia*. 2025. Available online: https://www.investopedia.com/terms/m/multilateral-netting.asp (accessed on 18 January 2026).
28. Tran, A.; Lu, Q.; Weber, I. Lorikeet: A model-driven engineering tool for blockchain-based business process execution and asset management. In *Business Process Management (BPM)*; Springer: Cham, Switzerland, 2018.

29. Loukil, F.; Boukadi, K.; Abed, M.; Ghedira-Guegan, C. Decentralized collaborative business process execution using blockchain. *World Wide Web* **2021**, *24*, 1645–1663. [CrossRef]
30. Di Ciccio, C.; Cecconi, A.; Mendling, J. Blockchain support for collaborative business processes. *Bus. Process Manag. J.* **2019**, *25*, 1–24. [CrossRef]
31. Possik, J.J.; Amrani, A.; D'Ambrogio, A.; Zacharewicz, G. A BPMN/HLA-based methodology for collaborative distributed discrete-event simulation. In *Proceedings of the 2019 IEEE 28th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*; IEEE: Piscataway, NJ, USA, 2019; pp. 118–123.
32. Meisami, S.; Bodell, W.E., III. A comprehensive survey of upgradeable smart contract patterns. *arXiv* **2023**, arXiv:2304.03405. [CrossRef]
33. Girish, N.M.; Kaganurmath, S. Upgradability of smart contracts: A review. *Int. Res. J. Eng. Technol.* **2022**, *9*, 2603–2606.
34. Palladino, S. The State of Smart Contract Upgrades. *OpenZeppelin Blog*. 2018. Available online: https://blog.openzeppelin.com/the-state-of-smart-contract-upgrades (accessed on 18 January 2026).
35. Rodler, M.; Li, W.; Karame, G.O.; Davi, L. EVMPatch: Timely and automated patching of Ethereum smart contracts. In *30th Usenix Security Symposium (USENIX Security 21)*; USENIX Association: Berkeley, CA, USA, 2021; pp. 1289–1306.
36. Li, Z.; Zhou, Y.; Guo, S.; Xiao, B. SolSaviour: A defending framework for deployed defective smart contracts. In *Proceedings of the 37th Annual Computer Security Applications Conference*; Association for Computing Machinery: New York, NY, USA, 2021; pp. 748–760. [CrossRef]
37. Corradini, F.; Marcelletti, A.; Morichetta, A.; Polini, A.; Re, B.; Tiezzi, F. Engineering trustable choreography-based systems using blockchain. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*; Association for Computing Machinery: New York, NY, USA, 2020; pp. 1470–1479. [CrossRef]
38. Corradini, F.; Marcelletti, A.; Morichetta, A.; Polini, A.; Re, B.; Tiezzi, F. Flexible execution of multi-party business processes on blockchain. In *Proceedings of the 5th International Workshop on Emerging Trends in Software Engineering for Blockchain*; Association for Computing Machinery: New York, NY, USA, 2022; pp. 25–32.
39. Falazi, G.; Hahn, M.; Breitenbücher, U.; Leymann, F. Modeling and execution of blockchain-aware business processes. *Softw.-Intensive Cyber-Phys. Syst.* **2019**, *34*, 105–116. [CrossRef]
40. Klinger, P.; Nguyen, L.; Bodendorf, F. Upgradeability concept for collaborative blockchain-based business process execution framework. In *Blockchain—ICBC*; Springer: Cham, Switzerland, 2020; pp. 127–141.
41. López-Pintado, L.; Mendling, J.; Weber, I. Dynamic role binding in blockchain-based business processes. In *Advanced Information Systems Engineering*; Springer: Cham, Switzerland, 2018; pp. 77–89.
42. Jin, J.; Yan, L.; Zou, Y.; Li, J.; Yu, Z. Research on smart contract verification and generation method based on BPMN. *Mathematics* **2014**, *12*, 2158. [CrossRef]
43. Werbach, K.; Cornell, N. Contracts ex machina. *Duke Law J.* **2017**, *67*, 313–382.
44. Financial Action Task Force. Updated Guidance for a Risk-Based Approach to Virtual Assets and Virtual Asset Service Providers. 2021. Available online: https://www.fatf-gafi.org (accessed on 18 January 2026).
45. Ferdous, M.S.; Chowdhury, F.; Alassafi, M.O. In search of self-sovereign identity leveraging blockchain technology. *IEEE Access* **2019**, *7*, 103059–103079. [CrossRef]
46. Mavridou, A.; Laszka, A. Tool demonstration: FSolidM for designing secure Ethereum smart contracts. In *Principles of Security and Trust (POST); Lecture Notes in Computer Science*; Springer: Cham, Switzerland, 2018; Volume 10804, pp. 217–227.
47. Mavridou, A.; Laszka, A.; Stachtiari, E.; Dubey, A. VeriSolid: Correct-by-design smart contracts for Ethereum. In *Financial Cryptography and Data Security (FC)*; Springer: Cham, Switzerland, 2019; pp. 446–465.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.