Automated Mechanism to Support Trade Transactions in Smart Contracts

Christian Liu
Faculty of Computer Science
Computer Science
Halifax, Canada
Chris.Liu@dal.ca[1]

Peter Bodorik
Faculty of Computer Science
Computer Science
Halifax, Canada
Peter.Bodorik@dal.ca

Dawn Jutla
Sobey School of Business
Saint Mary's University
Halifax, Canada
Dawn.Jutla@gmail.com

Abstract

Research on blockchains addresses challenges in developing smart contracts, especially for applications involving multiple actors and complex processes. Blockchain's constraints make creating these contracts difficult, particularly for long-term, multi-step trade activities. Although trade activities can be represented as smart-contract methods, blockchains lack adequate multi-actor transaction support. This paper reviews trade activity requirements and proposes an automated mechanism to generate smart contracts from Business Process Model and Notation (BPMN) models. It discusses modeling blockchain application requirements using BPMN and transforming these models into smart contracts for nested trade transactions. Real-life trade activities often require adjustments, necessitating smart contract repairs. We present a methodology for repairing contracts generated from BPMN transformations. Incomplete trade activities due to unexpected events trigger alternative arrangements in smart contracts. The repair process starts with the original BPMN model fragment, providing a modeler with the innermost transaction fragment containing the failed activity. The modeler amends the BPMN pattern based on successful previous activities. If repairs exceed the inner transaction's scope, they are addressed using the parent transaction's BPMN model. The amended BPMN model is then transformed into new smart contracts, ensuring consistent logic and data transitions. Our approach to transaction repair is demonstrated in generating smart contracts from BPMN models. Previously, we developed TABS+ as a Proof of Concept to transform BPMN models into smart contracts for nested transactions. This paper also discusses enhancing TABS+ to TABS+R, which supports transaction repairs.

Keywords— Blockchain, Smart Contracts, BPMN, Transaction Mechanism, Automated Generation, Repair Methodology, TABS+R

I. INTRODUCTION

The 2008 publication of the Bitcoin white paper and the launch of the Bitcoin blockchain in 2009 have sparked significant interest and research into blockchain technology. This technology has garnered attention from businesses, researchers, and the software industry due to its appealing characteristics like trust, immutability, availability, and transparency. However, like any emerging technology, blockchains and their smart contracts introduce new challenges, particularly concerning blockchain infrastructure and smart contract development.

Research is actively addressing several key issues, such as blockchain scalability, transaction throughput, and high costs. For instance, the high cost associated with consensus algorithms has been thoroughly studied, leading to the development and implementation of new consensus mechanisms. Additionally, challenges specific to smart contract development, such as limited stack space, the oracle problem (the blockchain's inability to interact with external data), data privacy, and compatibility across different blockchains, have also been explored in depth. Comprehensive literature reviews on these topics are available in various sources [1-10].

The constraints imposed by blockchains increase the complexity of smart contract development, especially for distributed collaborative applications. This complexity is highlighted by numerous literature surveys on the topic, such as those by Taylor (2019) [2], Khan (2021) [3], Vacca (2021) [4], Belchior (2021) [5], Saito (2016) [6], Garcia-Garcia (2020) [7], Lauster (2020) [8], and Levasseur (2021) [9]. To simplify smart contract development, research by López-Pintado (2019) [9, 11], Tran (2018) [12], Mendling (2018) [13], and Loukil (2021) [14] propose to express the requirements of a blockchain application using a model expressed in the Business Process Model and Notation (BPMN) that is then transformed into a smart contract.

Our research also starts with a BPMN model that is automatically transformed into smart contract methods, but our approach differs significantly as we use multi-modal Discrete Event (DE) – Hierarchical State Machine (HSM) modeling to transform the BPMN model into a DE-HSM model that allows for a graph-based representation of distributed blockchain applications, facilitating analysis and identifying patterns that remain isolated from other concurrent activities. We describe our approach in [15-22] together with a TABS tool (Tool to Automatically Transform a BPMN Model to Smart Contract Methods) as a Proof of Concept (PoC) we developed to show the feasibility of our approach.

[1] Corresponding author

In [22], we expand our approach from [15] to address collaborative activities in trade and distributed finance, to which we refer simply as trade activities. These activities often involve numerous steps and participants, such as price negotiations, letters of credit, transportation, and various documentation. Such trade transactions, involving multiple parties over extended periods, necessitate careful synchronization of activities across different software and human events. Furthermore, as transaction activities involve collaboration of participants, the trade activities are multi-step and thus require their synchronization. A smart contract must therefore represent these collaborative activities by invoking several methods.

However, a native blockchain transaction often falls short of representing these complex trade activities due to its focus on state changes rather than the collaborative nature of trade transactions. We use the term native blockchain transaction to refer to the general concept of a blockchain transaction. If a blockchain supports native cryptocurrency, we consider any transfer of a native cryptocurrency as a part of native blockchain transaction. The problem is that a trade transaction is naturally expressed as a collaboration of several methods that are invoked independently by the participants of trade activities, wherein a native blockchain transaction supports only the concept of a transaction as a change to the state of the ledger made by an execution of any of the methods of a smart contract, which of course may invoke other methods.

This mismatch is similar to the object-relational impedance mismatch [23]. We addressed this issue in our previous research [20] by proposing a methodology that allows developers to define a trade transaction as a collection of smart contract methods that can be invoked by different trade participants. We adapted database transactional properties (Atomicity, Consistency, Isolation, and Durability, or ACID) for trade transactions and incorporated access control and privacy features. Our approach uses pattern augmentation techniques to automate the creation of mechanisms that enforce these properties.

To develop a smart contract involving trade transactions, the developer writes the methods as usual, identifies the methods forming the trade transaction, and applies our methodology to integrate transactional properties. We also support nested trade transactions, imposing some restrictions to ensure that trade transaction methods only refer to objects and methods within their defined scope.

Since our initial proposal in [17], we have integrated nested trade transactions into our automated BPMN-to-smart-contract transformation project [22], exploring mechanisms to support transactional properties and their impact on access control, privacy, and recovery. Recognizing the importance of handling exceptions, we shifted our focus to automating recovery procedures for trade transactions, as smart contracts often encounter failure scenarios.

A trade transaction, which may be nested, representing a trade activity, involves execution of multiple methods of a smart contract, and thus recovery from a failure is more complex than recovery for a regular blockchain transaction. A recovery procedure needs to ensure that (i) the ledger is not affected a failed transaction, and (ii) that the applications, of different actors that participate in the trade transaction execution, be informed of the failures in correct sequence so that they can recover their resources on their local systems they dedicated to the execution of the failed trade transaction. In addition to invoking recovery procedures for the application, we also address the issue when the real-life situations prevent completion, and thus cause a failure, of a smart contract. In real-life, if trade activity arrangements cannot be completed due to some events/conditions, alternative arrangements are made. However, if such trade activities are modeled by a smart contract, the question is how to update/repair the smart contract to represent the trade activity repaired with new alternative arrangements.

In this paper, we present our initial approach to repairing trade (sub)transactions in smart contracts to reflect alternative arrangements. We utilize nested trade transactions to facilitate repairs, focusing on amending only the failed sub-transaction rather than the entire trade activity. We automate the generation and deployment of smart contracts and describe how to create and update versions of failed sub-transactions, ensuring continued execution of the smart contract post-repair.

A. Goal, Objectives, and Contributions

The primary goal of this paper is to present an approach enabling developers to declare a trade (sub)transaction as a collection of methods, allowing the system to automatically generate a transaction mechanism. This mechanism not only supports recovery from failure but also facilitates repair. If a failure occurs and the trade activity is subsequently repaired with alternative arrangements, the system ensures that the trade transaction representing the trade activity is also repaired. The specific objectives, which also constitute the contributions of this paper, are as follows:

- **Objective 1**: Describe the process for recovering a failed trade (sub)transaction to its state just before the transaction began. This recovery includes not only the restoration of the transaction state but also the invocation of recovery procedures for the distributed application, enabling it to recover local resources dedicated to the processing of the failed (sub)transaction.
- **Objective 2**: Investigate a methodology for trade (sub)transaction repair with the following considerations:
    - If possible, ensure that trade (sub)transactions representing successfully completed trade activities remain unaffected.

    - If possible, ensure that unexecuted trade (sub)transactions representing trade activities that follow the repaired/amended trade activity remain unaffected.
- **Objective 3**: Develop a Proof of Concept (PoC) demonstrating the feasibility of the proposed methodology for transaction repair.

B. Outline

The second section provides the necessary background, reviewing the desired transactional properties for trade (sub)transactions and relating them to the properties of transactions in databases and blockchains. As this research extends our previous work on the automated generation of smart contracts from BPMN models, the section also introduces BPMN models and overviews our approach to automating the generation of smart contracts from these models.

The third section details the recovery process for failed trade (sub)transactions, which restores the system's state to just before the transaction's invocation. This process must ensure that:

1. The ledger state remains unaffected by the failed (sub)transaction.
2. Recovery procedures for transaction participants are triggered to release local resources allocated for the failed (sub)transaction.

To address real-life scenarios where a trade activity failure, represented by a trade transaction, requires amendments, the fourth section describes our approach to repairing trade transactions based on the structure of nested trade transactions within a smart contract generated from a BPMN model. We explain how trade transactions are encapsulated in separate smart contracts and describe the process of repair by replacing the smart contract for the failed (sub)transaction with a revised version. The section also discusses the constraints and implications of such repairs on any preceding or subsequent activities related to the failed (sub)transaction.

The fifth section describes modifications made to our tool, TABS+, to create a tool TABS+R that supports transaction repair as a PoC. It discusses the potential benefits of supporting trade transaction repairs and identifies obstacles that need to be overcome for the broad adoption of our approach to the automated generation of smart contracts with repair capabilities.

The sixth section provides an overview of related work in the field and discusses limitations. The final section presents the conclusions of the study and describes future research directions.

II. BACKGROUND

This section provides an overview of BPMN modeling, introducing a simple BPMN use case utilized throughout the paper. We then discuss modeling with Finite State Machines (FSMs), Hierarchical State Machines (HSMs), and multi-modal modeling. We also review transactions in database (DB) and blockchain systems, comparing their properties with our concept of trade transactions. These concepts are foundational for our approach to repairing trade transactions generate from BPMN models.

A. Business Process Management Notation (BPMN)

BPMN, developed by the Object Management Group (OMG) [24-27], is designed to be comprehensible to a wide range of business users, from analysts to technical developers and managers. Its practical adoption is evident from various software platforms that enable modeling business applications with the goal of automatically generating executable applications from BPMN models. For example, the Camunda platform transforms BPMN models into Java applications [28], while Oracle Corporation converts BPMN models into executable process blueprints using the Business Process Execution Language (BPEL) [29].

Key features of BPMN models include flow elements representing computation flows between BPMN elements. A task represents computation executed when the flow reaches it. Other elements manage conditional forking and joining of computation flows, using Boolean expressions (guards) to control the flow or represent event handling. Additionally, data elements describe the data or objects flowing with the computations, serving as inputs for decision-making in guards or computation tasks.

B. FSMs, Hierarchical State Machines (HSMs), and Multi-modal Modeling

FSM modeling is widely used in software design and implementation, often extended with features like guards along FSM transitions. In the late 1980s, FSMs evolved into HSMs, incorporating hierarchical structures to facilitate pattern reuse, allowing states to contain other FSMs [30].

Girault et al. (1999) [31] describe combining HSM modeling with concurrency semantics from models like communicating sequential processes [32] and discrete events [33]. They describe how a system state can be represented by an HSM, with a

specific concurrency model, is applicable only to that state. This supports multi-modal modeling, where different hierarchical states can employ the most suitable concurrency models for concurrent activities in that state.

C. BPMN Model Transformation to Smart Contract Methods

In [15], we presented a methodology for transforming BPMN models into smart contracts. The transformation process involves several key steps:

1. **Transformation to Directed Acyclic Graph (DAG) Representation:** The pre-processed BPMN model is converted into a DAG representation. This mapping ensures that for any DAG vertex or edge, the corresponding BPMN element can be identified, and vice versa.
2. **Identification of SESE Subgraphs:** The DAG is analyzed to identify Single-Entry Single-Exit (SESE) subgraphs. Each subgraph has a Single Entry (SE) vertex, which is the only vertex of the subgraph that has incoming edges from vertices external to do subgraph, and a Single Exit (SE) vertex, which is the only vertex that has outgoing edges to vertices external to the subgraph. All other subgraph vertices have only edges connected to the internal nodes of that subgraph. SESE subgraphs are significant because once computation flow, as represented by the edges, enters a SESE subgraph, it remains confined within that subgraph until it exits via the exit node. This ensures a contained and manageable flow of computation.
3. **Transformation to DE-HSM Model:** The identified SESE subgraphs are used to transform the DAG representation into a Discrete Event Hierarchical State Machine (DE-HSM) model. Each node in the DE-HSM model represents either a DE-HSM sub-model or a computation expressed using concurrent FSMs, with some FSM states indicating execution of BPMN tasks.
4. **Elaboration and Flattening:** The DE-HSM model is further detailed, flattening the entire DAG into an interconnected network of DE-FSM (Discrete Event Finite State Machine) sub-models.
5. **Transformation to Smart Contracts:** The interconnected DE-FSM models are then transformed into smart contract code. Each BPMN task element is represented as a separate method within the smart contract. Task-method executions are triggered by specific state transitions in the FSMs, making the system's collaborative activities, i.e., the business logic, independent of the underlying blockchain infrastructure. Thus, the deployment of independently executed tasks can be managed separately from the blockchain layer.

It should be noted that if a smart contract method generates an event or raises an exception, due to the transformation process, the system is able to determine to which DE-FSM sub-models it belongs; then to which DE-HSM model it is a part of; and then it can determine which SESE subgraph of the DAG representation of the BPMN model, and hence also which part of the BPMN model it corresponds.

A smart contract is in essence an execution engine for concurrent FSMs. That is, each BPMN element representing a BPMN task computation is transformed into a separate method of a smart contract. A task-method execution is triggered when an FSM state is reached that indicates that on a transition to that state a particular task should be executed. What is important to note that the collaborative activities are represented by state changes in the concurrent FSMs and thus it is independent of the blockchain infrastructure. Thus, only execution of the independently executed BPMN tasks is blockchain dependent.

We exploited the concept of multi-modal modeling and independent subgraphs in [5] to support sidechain processing, in that the developer may choose to deploy a SESE subgraph as a separate transaction that is deployed on a sidechain. Thus, if a SESE subgraph has much computation to perform, such computation can be performed on a sidechain, albeit at the cost of overhead for communication between the mainchain and a sidechain. If computation performed on a sidechain is much cheaper than on the mainchain, then sidechain processing may be beneficial.

In [22], we use the nested structure of the SESE subgraphs to define nested trade transactions, which were initially introduced in [19], in the context of automated transformation of BPMN models into the methods of a smart contract(s). The BPMN model is transformed into a DAG and then into the DE-HSM model and developer is provided with information to decide on which of the SESE subgraphs are to be deployed as trade (sub)transactions, wherein the system automatically generates the transaction mechanism for each trade (sub)transaction. We facilitate options for a developer to select how the trade (sub)transactions should be packages and deployed, wherein one option packages and deploys each trade (sub)transaction as a separate smart contract. It is this option that is used to support repair of smart contracts.

III. RECOVERY PROCEDURES FOR NESTED TRADE TRANSACTIONS

We first describe the pattern augmentation scheme we use to generate the transaction mechanism automatically. We then discuss architectural alternatives available for the transactional mechanism, and then we proceed with evaluation of the alternative mechanisms.

In this section, we discuss the automated recovery procedures for nested trade transactions, focusing on restoring the system to the state just before the failed transaction. We will first outline the recovery procedures specific to blockchain transactions, followed by a discussion on how these procedures are facilitated within the trade transaction framework.

When an exception occurs during the execution of a smart contract method, the system checks for an associated exception handler. If the developer has provided an exception handler in the smart contract, it is invoked. This handler may resolve the issue, allowing the transaction to proceed without requiring recovery. However, if the exception cannot be resolved by the handler, the trade transaction fails, necessitating the execution of recovery procedures.

Blockchain transactions, including trade transactions, can fail due to exceptions during the execution of a smart contract or during the consensus phase, where the blockchain ensures consistency and serializability of transactions. The failure raises the question of whether the smart contract should be marked as failed and whether a new contract should be created to continue the trade activity, possibly utilizing some successfully executed parts of the failed contract. This context sets the stage for our approach to trade-transaction repair, detailed in subsequent sections.

As established, the trade transaction mechanism does not directly update the blockchain ledger. Instead, ledger writes are first cached and only committed during the trade-transaction commit phase upon successful completion. Therefore, if a blockchain transaction fails, ledger recovery is unnecessary. However, participants must be informed of the failure so they can release local resources allocated for the failed transaction's processing.

A. Recovery for Trade Transactions

The recovery procedure for native blockchain transactions is straightforward, as the blockchain infrastructure inherently ensures ACID properties. However, trade transactions involve multiple actors, each committing resources on their systems, complicating the recovery process. Figure 1 illustrates several actors participating in a trade transaction, each running their IT systems and potentially committing resources.

When a trade transaction fails, the recovery procedure must ensure:

1. The ledger state remains unaffected by the failed transaction's execution.
2. All participants are notified of the failure so they can release locally committed resources.

Since the trade transaction mechanism commits ledger updates only during the successful commit phase, there is no need for ledger recovery. However, recovery procedures for participant applications must follow the reverse order of the invocation of trade transaction methods. For nested trade transactions, this means that the recovery procedures of sub-transactions must be executed before those of the parent transaction.

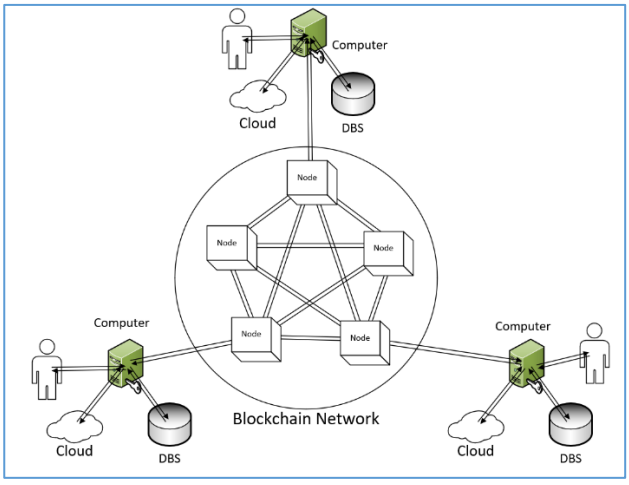

Figure 1. Blockchain Network and Participants

B. *Example Trade Activities and Nested Transations*

Consider a simple example of a smart contract that supports a trade transaction for a sale of a large product, such as a combine harvester. It may include price negotiation with payment via an escrow account, which is then followed by arranging transport. Transport arrangements include finding the requirements for the transport of the product, such as wide-load requirements or safety requirements in case of dangerous products in transport. Once the transport requirements are determined, the insurance and transport are arranged, and the product is shipped/transported. Following the transport, the product is received, and payments are completed. Figure 2 shows the trade activities as a BPMN model created using the Camunda platform invoked from our TABS+R tool that we describe in a later section. However, the model can also be viewed as a block diagram of the trade activities we use for exposition purposes.

These activities can be represented as separate trade (sub)transactions, such as:

- *priceAndEscrow_tx*: Includes price negotiation and escrow payment.
- *transportProduct_tx*: Divided into sub-transactions like *getDocs_tx* (determining transport requirements, obtaining insurance and transporter) and *doTransport_tx* (actual product transport).
- *receiveAndFinalize_tx*: Finalizes the transaction by receiving the product and completing payment.

Each trade (sub)transaction is encapsulated in a separate smart contract. If a trade transaction fails, recovery procedures must be executed in reverse order, starting with sub-transactions. Each recovery procedure notifies participants of the failure, enabling them to release their resources. Thus, three smart contracts are generated, one for each of the trade (sub)transactions priceAndEscrow_tx, transportProduct_tx, and receiveAndFinalize, wherein the trade transaction transportProduct_tx includes sub-transactions getDocs_Tx and doTransport_tx, as shown in Figure 3. In the figure, transactions are represented by dashed boxes with transaction names ending with the string "_tx", while activities/task(s) within a transaction are represented by dotted-line rectangles that were hand-drawn over Figure 2.

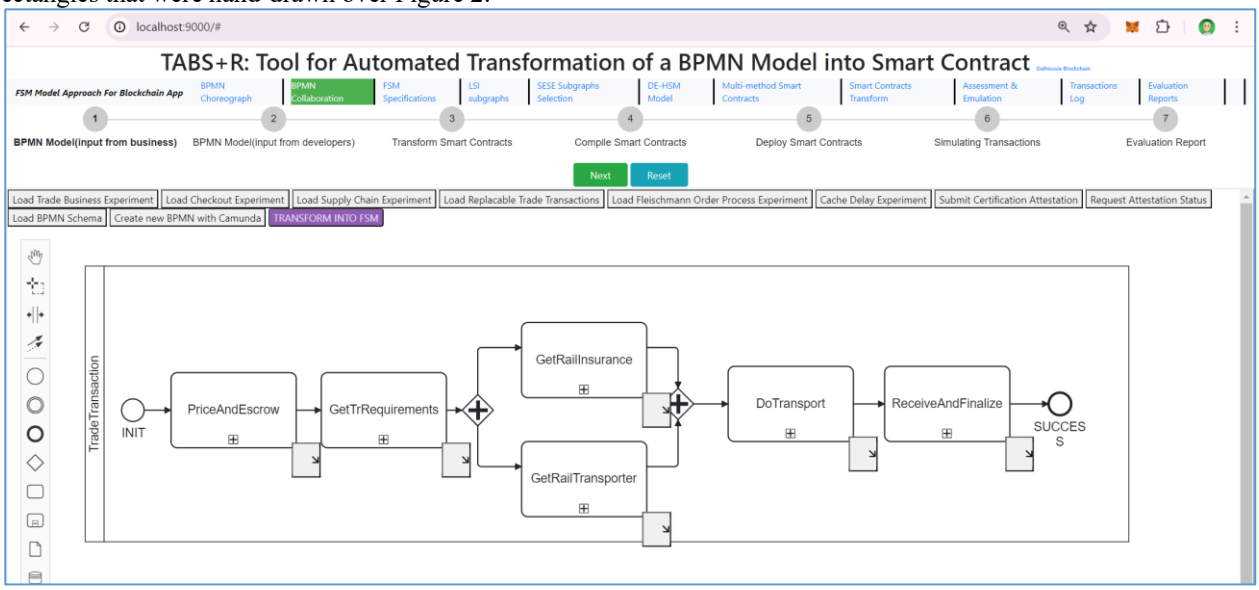

Figure 2. Block diagram of trade activities represented using a BPMN model created using Camunda platform [11]

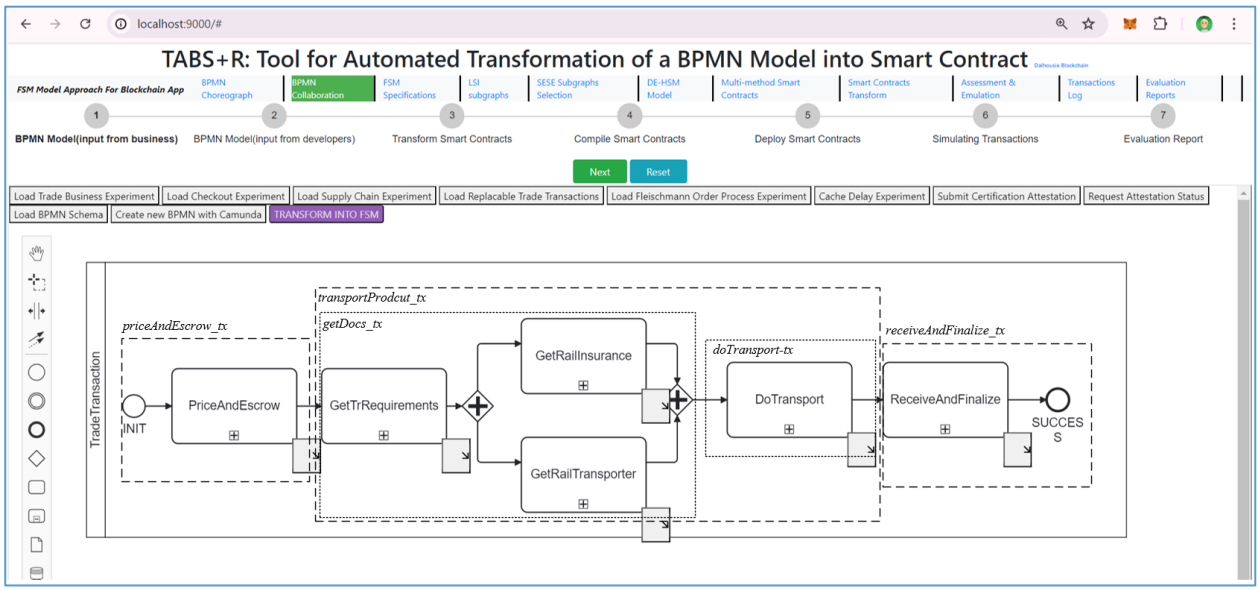

Figure 3. BPMN model of trade activities as nested trade transactions

In addition to the transactions shown in Figure 3, there is an additional smart contract, referred to as the main smart contract, that includes all non-trade-transaction methods that invoke the trade transaction methods. Furthermore, the methods of a sub-transaction are invoked from its parent transaction, while the methods of a trade transaction that is not a sub-transaction are invoked from the main contract that contains all methods that are not trade-transaction methods. In case of failure, recovery procedures for trade sub-transactions are invoked in the reverse order of the first invocation of their methods. Each recovery procedure for a trade transaction produces events to notify each of the transaction participants about the failure.

C. *Unanticipated Events and Completion of Transactions*

Developers often anticipate potential issues and write exception handlers to manage them, ensuring successful completion of trade activities. However, not all failures can be anticipated. For instance, a flood washing out a railway line might prevent product transport, a situation unlikely to have been foreseen by the developer.

In such cases, the question arises on how to complete the trade activity when part of the smart contract fails. Given the immutability of blockchain, representing new arrangements in the smart contract is challenging. One approach could involve creating a new smart contract tied to the failed one while leveraging successfully completed activities. Alternatively, a new contract could be derived from the failed one, incorporating completed activities. In the following section, we describe our approach to facilitating repair to complete the trade activity.

IV. TRADE-TRANSACTION REPAIR/UPGRADE

When a trade activity represented by a smart contract fails, the main objective is to amend or repair the smart contract to ensure the successful completion of the trade. Given that the smart contract is initially developed from a BPMN model of the trade activity, the repair process also involves creating a new BPMN model. This model represents alternative arrangements by modifying the original failed BPMN model, specifically replacing the pattern that caused the failure with one that includes alternative BPMN patterns that will not fail.

Our approach leverages the nested structure of trade transactions. Recovery from the failure of a trade (sub)transaction follows a well-defined process as outlined in the previous section. Notably, we package and deploy each trade (sub)transaction as a separate smart contract, localizing the repair. The repair can thus be achieved by upgrading or replacing the failed (sub)transaction with a corrected version.

Failure is first analyzed to determine a BPMN pattern that corresponds to the innermost trade transaction in which the failure occurred. Repair is attempted within the context of the BPMN model that corresponds to the trade transaction first. If the repair succeeds, then it is localized to the innermost transaction. If the developer is unable complete repair within the identified BPMN pattern that corresponds to the trade transaction, then repair of that pattern is aborted and, instead, repair is restarted, but this time within the context of the BPMN pattern of the parent trade transaction. Once the BPMN pattern is repaired, automated

transformation of a BPMN pattern into a smart contract is used to generate the smart contract representing the repaired BPMN pattern.

We first outline the repair of the failed BPMN pattern. We then describe generation of the smart contract for the repaired pattern, and how we achieve replacement, i.e., upgrade of the failed smart contract with the repaired version.

A. Repair at BPMN Model Level

The developer is presented with the original BPMN pattern that led to the failure, including the failure's cause. She/he must then replace this failing pattern with a new one that presumably avoids the failure. The repaired BPMN model integrates successfully completed activities from the failed smart contract's execution along with new elements to complete the trade activity. This process, though abstracted, is outlined in Figure 4.

The initial steps in Figure 4 involve determining which BPMN pattern within the model caused the failure. The cause of failure is often due to an unhandled exception or an exception handler failing to resolve the issue. Since failures occur during the execution of a smart contract method, translating this failure information from the smart contract context to the BPMN level is crucial. The developer uses this information to amend the BPMN pattern and repair the trade activity.

First, the failure information must be translated/mapped, from the context of a failed smart contract to the BPNM level representation, to provide the developer information about the BPMN pattern that needs to be amended/repaired and the reasons for failure. The user is then presented with a BPMN pattern to be repaired and information about that pattern, including information flowing in and out of the pattern, purpose, and cause of the failure. The developer repairs/replaces the BPMN pattern causing the failure with a repaired BPMN pattern that is transformed into the methods of a smart contract that replaces its failed version.

1. **BPMN Model Failure Information:** Information about the failure is gathered, identifying the BPMN pattern that caused it. The repair begins with the BPMN pattern associated with the innermost trade (sub)transaction where the failure occurred.
2. **Model Amendment:** The developer is shown the original failing BPMN pattern (Pf), including details on the reason for failure, the pattern's intended function, and the objects involved. The goal is to replace Pf with a repaired pattern (Pr) under the following constraints:
   - The computation in the new pattern should use the same objects that were inputs to the failed execution.
   - The output objects from the new pattern must include, at a minimum, those produced by the failed computation.

   If the pattern cannot be amended while satisfying these constraints, the repair escalates to the parent trade transaction's BPMN pattern.
3. **Smart Contract Generation:** Upon completing the BPMN model's repair, the system generates a new smart contract from the updated BPMN model of the pattern.

Figure 4. Repair Steps

Consider a scenario where the *doTransport_tx* fails due to a washed-out rail line. The BPMN model shows that insurance and a transporter had been arranged, but the transport could not occur. If an alternative route is available with the same transporter and insurance, the constraints are satisfied, and the repair remains within the *doTransport_tx* context. However, if the transport must switch to a road route with different insurance and transporter, the repair must extend to previous trade transactions *getInsurance_tx* and *getTransporter_tx*. If the repaired pattern does not meet the required outputs for subsequent activities, the repair extends to these activities, as outlined in Figure 4. The final step involves generating a new version of the smart contract from the repaired BPMN model. Of course, before the repair of the smart contract can proceed, alternative transport arrangements need to have been discovered and arranged, as the BPMN model and the generated smart contract only represent/model execution of such activities.

Continuing with our example, assume an alternative arrangement is found for the product, with the same transporter using an alternative rail-line route, and existing insurance applies to the new route. Since the alternative route arrangements are performed by the same transporter and the insurance covers the transport via the alternative route, the already completed activities performed by the *getInsurance_tx* and *getTransporter_tx* sub-transactions do not need amendment/repair. This is the case when constraint (a) of step 2 in the repair outline shown in Figure 4 is satisfied.

However, consider a situation where *doTransport_tx* fails, and no other rail route is available, necessitating road transport. Assume the hired rail transporter does not provide road transport, and the insurance for rail transport does not apply to road transport. This situation fails to meet constraint (a) in step 2. The previously completed trade activities of obtaining insurance and arranging transport must also be repaired. In such a situation, the repair within the context of the trade transaction is aborted, and the repair of the previously completed activities must be amended/repaired to accommodate alternative transport arrangements. In such a situation, the repair within the context of the trade transaction is aborted, and instead, the repair of the previously completed activities needs to be amended/repaired to provide appropriate information for the alternative transport arrangements.

Thus, the repair restarts for the BPMN pattern representing the parent trade transaction, *productTransport_tx*, of the failed *doTransport_tx* sub-transaction. Since the *productTransport_tx* parent transaction includes the *getInsurance_tx*, *getTransporter_tx*, and failed *transportProduct_tx* sub-transactions, the developer must amend the BPMN pattern, including the *getInsurance* and *getTransporter* trade activities, represented by the *getInsurance_tx* and *getTransporter_tx* sub-transactions, respectively.

When the computation flow exits the repaired BPMN pattern, it must produce information required by succeeding trade activities. If the repaired pattern produces all necessary information, the succeeding activities are unaffected and need no amendment/repair. However, if the repaired pattern does not provide all required information, repair extends to the parent trade (sub)transaction. This represents the situation where constraint (b) of step 2 in Figure 4 is not satisfied. The final step uses the amended/repaired BPMN model to generate smart contract methods for the repaired BPMN model.

In the following subsections, we elaborate on how the repair steps are accomplished within the context of automated smart contract generation from BPMN models. We also discuss issues that need to be addressed to bring automated generation of smart contracts from BPMN models closer to adoption for supporting trade activities. The next subsection describes a proof of concept (PoC) created to test the feasibility of our repair approach.

B. Repair: From BPMN Model to Smart Contract

In this subsection, we detail each repair process step shown in Figure 4, focusing on transformations/mappings between BPMN and smart contract abstraction levels.

1) BPMN Model Failure Information

Recall that the process of generating a smart contract from a BPMN model starts with analyzing the DAG representation of the BPMN model to find SESE subgraphs of the DE-HSM model, which the developer uses to define nested trade transactions. The DE-HSM model transformation into smart contract methods involves packaging and deploying each trade (sub)transaction as a separate smart contract with its methods.

Also, recall that failure is detected during the original smart contract's execution when an unhandled exception is raised. Since a trade (sub)transaction is defined using a SESE subgraph, identifying the smart contract where the exception was raised is straightforward when a failure occurs. The exception is raised in a method belonging to a trade transaction represented by a subgraph in a DE-HSM model built from a BPMN model's DAG representation. Consequently, the repair system can determine the BPMN pattern needing repair, corresponding to the SESE subgraph representing the trade transaction where the failure occurred.

2) Model Amendment

After presenting the BPMN model and failure information to the developer, she/he attempts to repair the identified, failed BPMN pattern to generate a new, repaired BPMN pattern. The repaired BPMN pattern must ensure the repair is limited to the failed pattern, maintaining the DAG representation of the repaired BPMN pattern as a SESE subgraph. The repair must satisfy the following constraints:

a. Information flowing into the repaired BPMN pattern Pr and its corresponding subgraph must match or be a subset of the information flowing into the original BPMN pattern Pf.

b. Information flowing out of the repaired BPMN pattern Pr and its corresponding SESE subgraph must match or be a superset of the information flowing out of the failed BPMN pattern Pf.

These constraints ensure that the effects of executing the repaired BPMN pattern and its corresponding SESE subgraph and trade (sub)transaction are localized, not affecting preceding or succeeding trading activities.

It must be ensured that information required for executing the pattern is available, produced by previously completed activities, as per constraint (a) in Step 2 of Figure 4. In our example, if the same transporter and insurance can be used for the alternate transport route, the repair can be accomplished within the doTransport_tx transaction context. The developer performing the repair determines whether the existing insurance and transporter are applicable for the repaired activities. If a new transporter and/or insurance are required, the repair process restarts for the parent transaction.

Constraint (b) in Figure 4 ensures the repaired pattern's computation produces information required for subsequent computations. This means the objects flowing out of the repaired BPMN pattern Pr must include those flowing out of the failed pattern Pf.

3) Repaired Smart Contract Generation

Each trade (sub)transaction is packaged and deployed in a separate smart contract. To facilitate replacing a failed smart contract with a repaired one generated from the repaired BPMN pattern with alternative transport arrangements, two tasks must be accomplished:

i. A new version of the smart contract generated from the repaired BPMN pattern needs creation and deployment.

ii. Invocation of the new version of the smart contract, representing the repaired pattern Pr, must replace invocation of the original smart contract representing the failed BPMN pattern Pf.

For (i), the developer creates a new BPMN pattern for the failed activity, not the entire trade activity.

For (ii), the architecture of the execution model for smart contracts generated from BPMN models is exploited. As described, applications do not directly invoke smart-contract methods; instead, they invoke an API prepared by the TABS+ tool, which in turn invokes the smart contract methods. To "upgrade/repair" a trade (sub)transaction like *productTransport_tx*, the API is updated to point to the new smart contract version and its methods, ensuring the invocation of the new *productTransport_tx* transaction version instead of the failed version.

While the concept of transaction repair has been presented, many details need to be worked out. We are investigating security issues with this approach to prevent subversion and exploring secure approaches to upgrading smart contracts. The discussed trade-transaction repair concept requires further details, particularly from a security perspective.

Before repairing a BPMN pattern, alternative arrangements must be found by a business analyst, a process outside this paper's scope. However, once such arrangements are made, their inclusion in the BPMN pattern under repair is assisted by the repair system, providing the developer with information on the objects flowing into and out of the computation performed by a trade (sub)transaction. We are progressing toward supporting smart contract generation for trade transactions and creating infrastructure for the automated generation and repair of trade transactions in trade of goods and services. The next section describes the TABS+R tool developed as a PoC to demonstrate the feasibility of our approach.

V. PROOF OF CONCEPT: TABS+R TOOL

In our previous work [15], we demonstrated the feasibility of transforming BPMN models into smart contracts using the TABS tool. Subsequently, we introduced the concept of nested trade-transactions in smart contracts to support complex collaborative activities beyond the capabilities of native blockchain transactions [22]. We extended the TABS tool into TABS+ to facilitate the automatic generation of smart contracts that implement these nested trade transactions and their transactional properties.

To evaluate the feasibility of smart contract repair, we further developed TABS+ into TABS+R. This augmented tool supports the repair of smart contracts using the approach detailed in the preceding sections. We provide an overview of the tool's interface and outline the main steps involved in repairing a smart contract.

It should be noted that the TABS+R tool is configured for research, experimentation, and testing. It includes features and steps not intended for production environments, such as stepping through the transformation process and inspecting inputs and outputs. It also measures various delays by capturing timing at different points.

We first describe the process of transforming a BPMN model into smart contract methods using TABS+R, using the repair scenario from Figure 3 as a case study. Next, we explain how the tool assists developers in facilitating repairs and resuming trade activities.

A. TABS+R Tool Overview

TABS+R utilizes the Camunda platform [28] for creating BPMN models according to BPMN specifications [26-27]. These models are stored in XML format. Figure 2 displays a partial screen of TABS+R during BPMN model creation for our example application. Once the BPMN model is saved in XML format, it is transformed through a series of steps controlled via tabs in the tool's interface. The initial tabs focus on BPMN modeling, while subsequent tabs manage the transformation steps until the generation of smart contracts.

After creating the BPMN model, the developer guides its transformation into smart contracts by interacting with the tool, providing template methods for BPMN tasks. The tool supports generating smart contracts for Hyperledger Fabric blockchains

or EVM-based blockchains, such as Quorum or Ethereum. The mainchain smart contract can invoke methods of smart contract (s)deployed on a sidechain(s).

Figure 5 shows a screenshot of the transformed DE-HSM model with SESE subgraphs derived from the BPMN model of trade activities in Figure 3. The left-hand panel displays the BPMN graph and SESE subgraphs identified by TABS+R. The right-hand panel lists these subgraphs as selectable boxes for the developer to define trade (sub)transactions.

The developer decides which independent subgraph patterns should be deployed on a sidechain as separate smart contracts interacting with the main contract. The choice between Ethereum and Hyperledger Fabric for the mainchain and sidechain is made by the developer also. For testing, local blockchains are set up for both Ethereum and Hyperledger Fabric, with prepared channels on Hyperledger Fabric for smart contract deployment. For Ethereum-compatible sidechain, Quorum is used. After the selections, the model is transformed into methods of smart contract(s) that are then deployed and executed.

The developer can step through the generated system's execution, observing message-by-message progress. The tool graphically represents state changes in individual FSMs of the DE-FSM sub-models (Figure 6), aiding in testing and manual verification. Execution delays are displayed as the process proceeds. Additionally, developers can generate exceptions for specific activities, which is useful for testing and evaluating transaction repair. In Figure 6, fault exception propagation is highlighted in red.

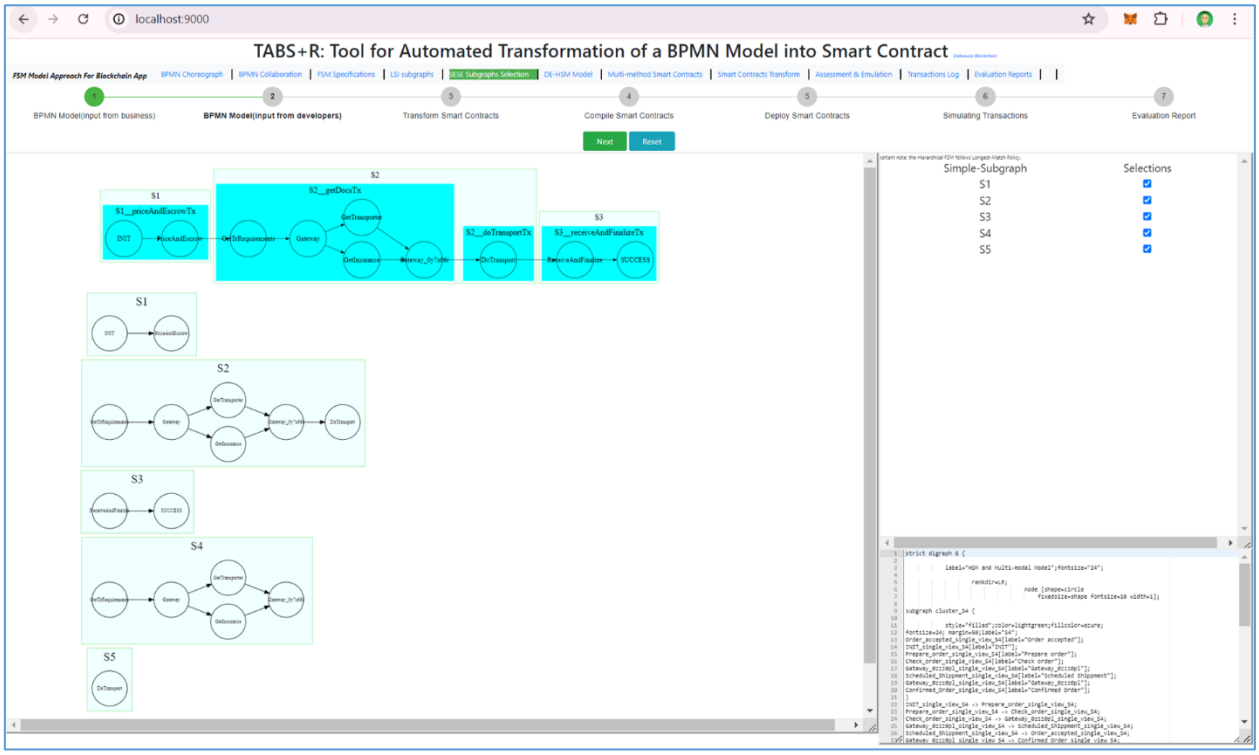

Figure 5. Identified SESE subgraphs and their selection

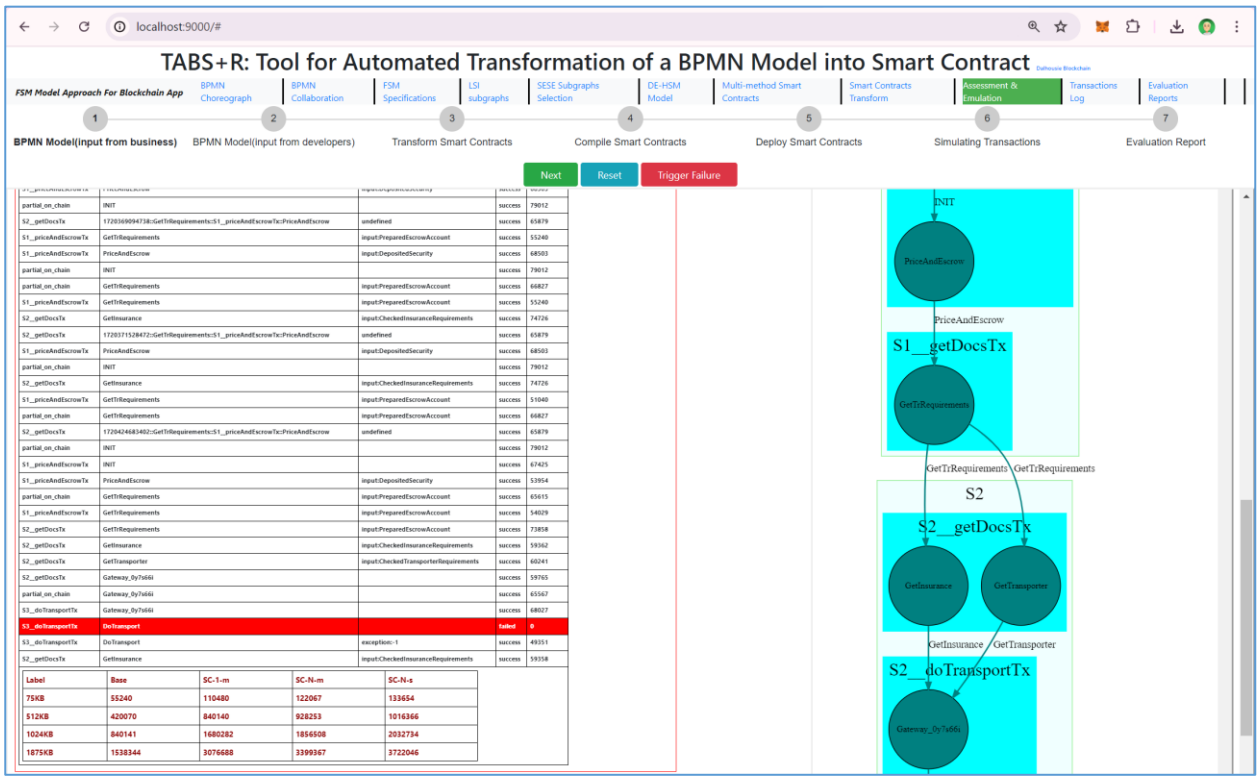

Figure 6. Monitoring execution of trade (sub)transactions and their activities for evaluation purposes

B. Repair of Trade (Sub)Trnasactions with TABS+R

When an unhandled transaction failure occurs during the execution of a trade (sub)transaction, TABS+R presents the developer with a popup BPMN model diagram corresponding to the failed trade activity. The developer then amends the BPMN model to create a repaired version that resolves the failure (see Figure 7).

In the repair process, the developer modifies the BPMN model of the failed trade (sub)transaction while ensuring that: (a) the input data flowing into the sub-transaction, and (b) the output information flowing out, remain consistent with the previous, failed version. For example, if the internal transport activity, represented by the BPMN task DoTransport, initially involves transport by rail and needs to be amended to transport by road, the essential inputs and outputs should not change. If the developer can successfully amend the activity to repair the sub-transaction by adjusting the BPMN task DoTransport, they then use the tool to transform the repaired model into a smart contract. This new smart contract is deployed on the same blockchain as the original failed version. Additionally, the Dapp's API must be updated to invoke the repaired sub-transaction instead of the failed one. Once these changes are made, the developer instructs the tool to proceed with the execution of the repaired sub-transaction and continues with the workflow.

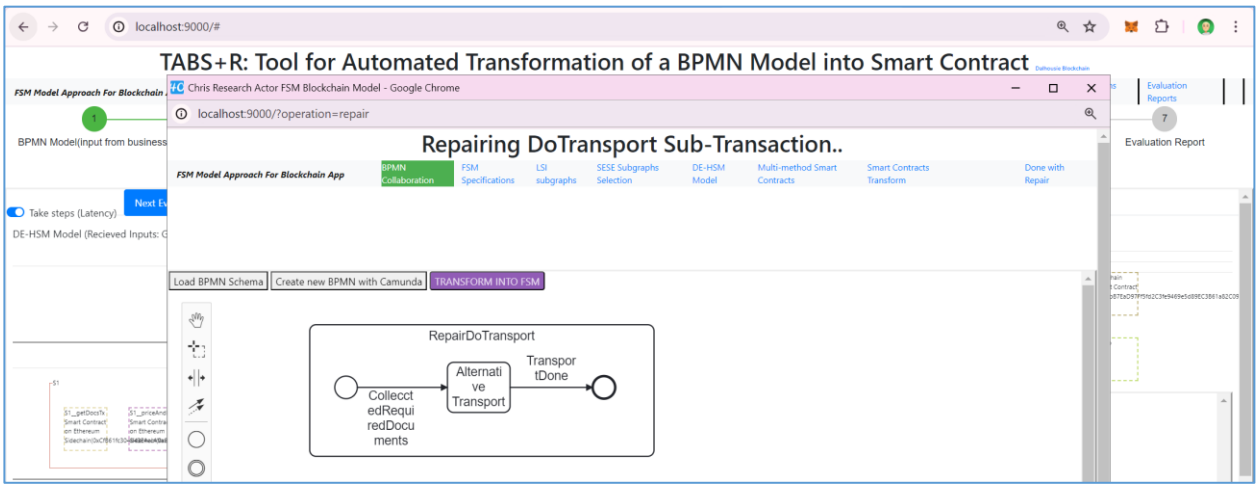

Figure 7. Repair of the trade sub-transaction doTransport_tx

The achieved repair (Figure 7) is contingent upon the applicability of input information, such as insurance documents and transporter selection, to the new transport method. Specifically, the same transporter must be able to provide road transport, and the same insurance must cover the new mode of transport. If these conditions are met, the repair of the doTransport_tx sub-transaction proceeds as planned. However, if either condition is not met, the repair of the sub-transaction is aborted. In such cases, the developer must repair the parent transaction instead, as shown in Figure 8. This process involves arranging new insurance and transport by road, which need to be recorded in the smart contract.

After completing the BPMN model for the repair, the developer instructs the tool to transform the repaired model into a smart contract and deploy it. Finally, the developer directs the tool to execute the repaired sub-transaction and continue the overall workflow.

C. *Tool Evaluation and Developer Feedback*

Although the primary goal is to gain acceptance of the TABS+R tool among developers, the current version is not yet ready for formal evaluation. This is due to the tool's limitations and its interface, which is currently geared toward design and testing rather than production use. Additionally, attracting developers to test the tool without compensation has been challenging. However, informal demonstrations to a blockchain company developers have elicited favorable feedback and suggestions for improvement, indicating a positive reception of the tool and its approach. Future work will focus on overcoming existing limitations and enhancing the user interface according to UI and UX guidelines.

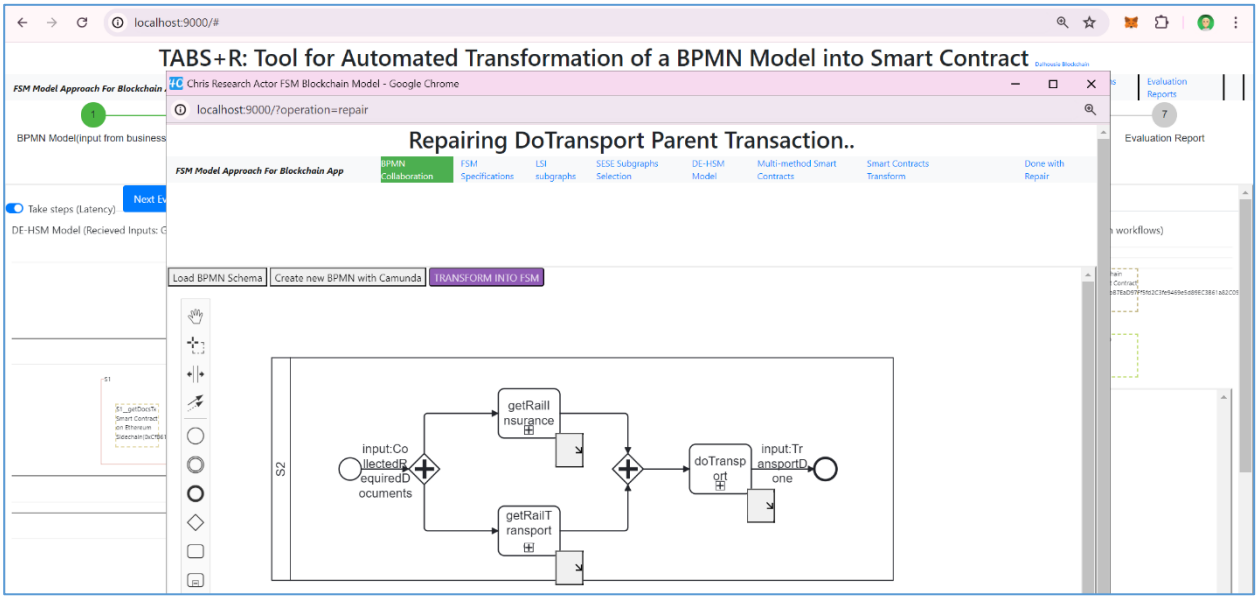

Figure 8. Repair of the trade transaction transportProduct_tx, parent transaction of the failed doTransport_tx

VI. RELATED WORK, LIMITATIONS, AND FUTURE WORK

A. Related Work

The Lorikeet project [12] uses a 2-phase approach to transforming BPMN models into smart contracts. In the first phase, the BPMN model is analyzed and transformed into smart contract methods, which are then deployed and executed on a blockchain, specifically Ethereum. An off-chain component facilitates communication with the Dapp. The actors exchange messages according to the BPMN model, with these exchanges being managed by the off-chain component. The smart contract includes a monitor that stores and enforces the model choreography, ensuring that message exchanges adhere to the predefined sequence. In addition, the project provides support for asset control, including both fungible and non-fungible assets. It provides a registry of tokens and methods for asset management, such as transfers, enabling rapid prototyping of smart contracts from BPMN models. This allows for the creation, testing, and modification of smart contracts before deployment.

Caterpillar [11,34] offers a different approach, focusing on BPMN models within a single pool, which is a BPMN construct, where all business processes are recorded on the blockchain. Its architecture comprises three layers: Web Portal, Off-chain Runtime, and On-chain Runtime. The On-chain Runtime layer includes smart contracts for workflow control, interaction management, configuration, and process management, with Ethereum as the preferred blockchain.

Loukil et al. (2021) [14] introduced CoBuP, a collaborative business process execution architecture on blockchain. Unlike other approaches, CoBuP does not compile BPMN models directly into smart contracts but instead deploys a generic smart contract that invokes predefined functions. It features a three-layer architecture: Conceptual, Data, and Flow layers, transforming BPMN models into a JSON Workflow model. This model governs the execution of process instances, which interact with data structures on the blockchain.

Di Ciccio et al. (2019) [35] compared the approaches of Lorikeet and Caterpillar across several features, including model execution, BPMN element coverage, incorrect behavior discovery, sequence enforcement, participant selection, access control, and asset control. They highlighted the unique aspects of each approach and provided a basis for comparison.

In [22], we expanded the comparison performed in DiCiccio by inclusion of CoBuP [14] and TABS+ approaches and adding the following features for comparison purposes: support for nested transactions, deployment capabilities, type of synchronization, and privacy. Features that distinguish our approach and the TABS+ tool from others include the transformation of a BPMN into a DAG representation and then to DE-HSM and DE-FSM models that enable the analysis of process flow to identify localized processing using SESE subgraphs. Unlike other systems that use direct translation of BPMN models into smart contract code, TABS+ produces smart contracts that are abstract representations of process flows expressed through concurrent FSMs. The logic of the process is executed by a smart contract executing the firings of concurrent FSMs, while some of the transitions cause executions of localized BPMN tasks, each one executed by a smart contract method with well-defined inputs and outputs. Furthermore, localized SESE graphs may be packaged and deployed as separate smart contracts that may be deployed on sidechains or may be used to define nested trade-transactions with well-defined transactional properties.

Similar to CoBuP, Bagozi et al. [36] use a three-layer as is but simpler approach. In the first layer, the collaborative process is represented in BPMN by a business analyst. In the second layer, a business expert adds annotations to the BPMN model that identify trust-demanding objects, and then Abstract Smart Contracts, which are independent from any blockchain technology, are created. Finally, Concrete Smart Contracts are created and deployed on a specific blockchain.

Mavridou and Lazska [37, 38] advocate correct design of smart contracts. They target systems that have their requirements represented using an FSM that is transformed into the methods of a smart contract. Transitions may result in execution of specific tasks that are represented as methods of smart contracts. Then, each method of the smart contract is analyzed and patched for any security holes. They develop a tool, called FSolidM, as a PoC that examines each smart contract method and augments it with security code to eliminate discovered security issues. Furthermore, after hardening a smart contract method, steps are taken to prevent the developer from modifying the inserted security code when amending the method's functionality before the smart contract deployment.

Mavridou and Lazska, together with Stachtiari and Dubey, advance the work on FSolidM by creating a framework, VeriSolid [39]. The smart contract is specified using a graphically specified transition system, which is an FSM extended with variable declarations and guards expressed in the Solidity language, plus a list of verifiable properties. The transition system is analyzed using the declarations and guards and transformed into a transition system with augmented states to ensure that the

desirable properties are satisfied. Only then is the system transformed into the methods of a smart contract expressed in Solidity, resulting in a correct design in terms of satisfying the expressed properties.

Verification of smart contracts generated from BPMN has been addressed in [40]. They use a two-step transformation process. In the first phase, the BPMN model is transformed to Prolog that is used to validate the business logic. Following this, they transform the BPMN model into a smart contract in GO language. However, there is no validation of the generated code.

In our approach, once the BPMN model is transformed into the DE-FSM model, it represents the business logic using concurrent FSMs, which is a transition-based system. Similar to FSolidM, we check the methods written by the developer to represent BPMN tasks for known security bugs and prevent them. However, we do not yet support representation of desirable properties in the BPMN model.

When smart contracts are used to represent collaborative activities of participants, it has been acknowledged that the blockchains immutability property is desirable for promoting trust, but also causes difficulties as frequently such collaborations need to be augmented to either repair security issues, or to support new desirable features. Thus, upgradability of smart contracts is an important issue that has been tackled in literature, particularly in the context of security of smart contracts.

For instance, Rodler et al. [41] describe a framework, EVMPatch, that uses a bytecode rewriting technique to automatically rewrite common off-the-shelf contracts to upgradable contracts. EVMPatch upgrades faulty Ethereum smart contracts using a bytecode rewriting to patch common bugs, such as integer over/underflows and access control errors.

Another bug fixing framework is SolSaviour [42] that uses a voteDestruct mechanism to allow contract stake holders to decide to withdraw inside assets and destroy the defective contract, which is followed by repairing and redeploying the smart contract while incorporating the old contracts' internal states to the new ones. Our tool TABS+R is similar in that the repaired version of the smart contract incorporates the successfully completed activities of its unrepaired version.

Aroc [43] is yet another framework for repairing smart contracts written in Solidity for an EVM with the objective of not modifying the vulnerable smart contract itself. That is, instead of the using a proxy contract that invokes a patched/repaired and redeployed version of the vulnerable smart contract, the authors propose that a new smart contract is created that prevents attacks on the faulty contract. The Aroc system first generates and deploys a patch smart contract that blocks invocations of the vulnerable smart contract by malicious smart contracts that try to exploit the vulnerability. The owner of the vulnerable smart contract uses a special transaction supported by an extended EVM that supports the re-direction of the invocations of the vulnerable smart contract to the patch smart contract.

Corradini et al. [44, 45] address the tension between the trust in blockchain's trust, achieved via immutability, and the need for flexibility required for multiparty collaboration. They use BPMN to model business processes, which are then transformed into code, while it is the code's execution state that is stored on the blockchain. This decoupling of business process choreography from its execution state allows for run-time changes to the process execution. They developed a tool, FlexChain, to show feasibility of their approach.

Falazi et al. [46] addressed the issue of using smart contracts in support of business processes modeling and developed a prototype, BlockME, to validate their approach. They use BPMN to represent the choreography of the business processes while extending BPMN task to support invocation of smart contract methods for permissionless blockchains. The BPMN model is transformed into Business Process Execution Language (BPEL) for execution. Choreography of process is executed via BPEL that invokes smart contract methods that implement the BPMN task elements. Authors extend their approach and BlockME prototype in [47] to support invocation of smart contract functions of different blockchains, including both permissioned and permissionless by developing a new technique to identify smart contract functions and a metric to gauge transaction finality. In short, the collaboration logic is executed in BPEL and not smart contract methods as in our TABS+ approach.

The closest work to our approach in repairing/upgrading smart contracts is [48]. The paper analyzes and implements three different upgradeability concepts, one based on a registry, one using a proxy pattern, and the third one based on a registry combined with a pattern segregation. Their use case is a large organization with departments. The BPMN model of the collaboration is transformed into smart contracts, with each department's activities represented by a smart contract. Upgrade thus needs to be carefully managed to ensure that activities already executed are consistent in the context of the upgrade. Their findings suggest the Unstructured Storage Proxy pattern to be the most promising for practical use, especially regarding cost-effectiveness and minimal added complexity.

In comparison to [48], our TABS+R approach naturally packages activities into different smart contracts, wherein our use of nested trade transactions is exploited to protect against inconsistencies due to some of the activities being completed by the old version of the smart contract while some are executed with the new repaired/upgraded smart contract.

To the best of our knowledge, we are not aware of any formal work on multi-method transactions for blockchain smart contracts to which we refer as trade transactions, wherein a trade transaction contains executions of independently invoked smart contract methods by different actors. We introduced the concept of a multi-method transaction in [15, 22], and in this paper we exploited it to repair/upgrade smart contracts in the context of automated generation of smart contract from BPMN models.

B. *Limitations and Future Work*

Although we feel that our approach to ease the developer's task in creating smart contracts for trade transactions is progressing, there are still many problems and limitations that need to be addressed. In this subsection we describe both the limitations and our plans on how to address them.

Securing Smart Contract Methods

To secure smart contracts, we adopted the approach in [37-38], wherein they propose hardening of smart contracts created by transformation of an FSM to smart contract methods. Given an FSM as a representation of the smart contract activities, they proposed a transformation of the FSM into the methods of a smart contract. They then propose securing each of the smart contract methods by inserting security patterns to guard it against (i) reentrancy by using locking, (ii) transaction ordering in face of unpredictable states, (iii) timed transitions, and (iv) access control. We successfully incorporated into the TABS+ the reentrancy protection by inserting appropriate locking patterns into the smart contract and support access control. In essence, the security patterns are inserted into the smart contract method at the start of a method and at its end. However, in our future work we shall develop smart contract patterns to guard against all known smart contract vulnerabilities. In addition, unlike native blockchain transactions, for trade transactions we also need to protect against Main-in-the-middle attacks.

Validation and Verification

Validation and verification need to be an integral part of the transformation process from BPMN models to smart contracts deployment. Transformations of BPMN models results in a DE-FSM model that is a transition system, and we plan on applying the VeriSolid [39] verification methods to ensure that generated smart contracts are correct by design. We already ask the BPMN modeler to document information flowing along with the execution flow. We shall extend that documentation also with the desirable properties and then perform formal verification of the system in terms of liveliness, reachability, deadlock-free properties and that the desirable properties are satisfied.

Blockchain Agnostic Smart Contracts

One of our objectives is to achieve generation of smart contracts that are blockchain agnostic. We made progress towards this objective as the collaboration logic is blockchain independent as it is expressed in terms of interconnection of the DE-FSM models, and it uses DEs to model concurrency and concurrent FSMs to model the functionality. However, currently, to apply a smart contract developed for one blockchain to be deployable and executable on another blockchain, the scripts for the BPMN task elements are provided by the developer and need to be executable on the target blockchain. To overcome this issue, we are investigating a two-layer approach taken by the Plasma project, described in [49], in which the task scripts are not executed on the blockchain, but rather off-chain, while the smart contract simply guides the collaborations and obtains certifications about the results of the tasks that are executed off chain.

VII. SUMMARY AND CONCLUSIONS

Trade of goods and services, including distributed finance, contains activities that require specialized customization that is not yet easy to support by traditional development of smart contracts. We provided motivation and need for a concept of a trade transaction as a collection of methods of a smart contract, and we derived the transactional properties for trade transactions that are based on the ACID transactional properties of transactions in DB systems and native blockchain transactions, augmented with two optional properties of access control and privacy.

We then described how a pattern augmentation approach, to automatically create transaction mechanism for nested trade transactions, is incorporated into the generation of smart contracts from BPM models. Furthermore, we also showed how a transaction mechanism is augmented to recover a failed transaction so that not only the ledger is unaffected by a failed transaction, but also that actors participating in the failed trade transaction are informed of the failure so that they could perform recovery on their systems to release resources committed to now failed trade transaction.

Activities in the trade of goods and services are subject to effects by external events that may cause failure of a trade activity supported by a smart contract. Such a failure thus precludes successful completion of a smart contract unless that failed activity can be replaced with one that will succeed and facilitate successful completion of the trade activity. We call such an amendment/upgrade of a smart contract as a transaction repair, although referring to it as a transaction upgrade or replacement would server as well. We described how we use the concept of nested trade transactions, together with automatically generated transaction mechanism, to support the repair/upgrade of a failed smart contract so that it can be completed. The repair process exploits the concepts of the nested trade transactions to ensure that the successfully completed activities of the smart contract

failed version may be incorporated consistently into the new version of the smart contract modeling the alternative trade activities to avoid failure.

However, when a trade activity needs to be amended/repaired to respond to external situations, such repairs are performed by business analysts who can then repair the BPMN model representing the activity. However, as the trade activity is represented by a smart contract, replacing an activity in a smart contract requires an effort that is equivalent to writing a smart contract in the first place. This is because the developer time needs to be allocated to the activity and the developer must familiarize herself/himself with the requirements of the contract and the changes that are required; write and test the amended to the smart contract; and deploy the smart contract.

As can be appreciated, delays with allocating a developer's time, and developer's time needed to write the new version of the smart contract representing a failed transactions are too long, particularly in situations when the smart contract representing trade activities need to be made promptly, such as in the use case described in this paper. Thus, we strive for fully automated generation of smart contracts and their repair that can be under the control of a business analyst only, without assistance of a software developer.

There are two obstacles in that effort. Software developer is required for the selection of which SESE subgraphs should be deployed as trade transactions. We are eliminating this step by automatically deploying all SESE subgraphs as trade (sub)transactions by default. In addition to not requiring the developers input, this should also provide more options when repairing transactions in case repair is not possible within an inner sub-transaction. The major obstacle, however, is in generating methods that implement the PBMN task elements. We are investigating if there is some simple language/model to represent the BPMN task functionality that can be automatically translated into a smart contract method for the target blockchain.

By supporting automated creation of smart contracts from BPMN models and providing support for augmentation of BPMN models with BPMN patterns and replacement of patterns in BPMN models with similar patterns, we are striving to create an environment to provide a relatively new concept of Smart-Contract-as-a-Service (SC-as-a-service). In short, a modeler would be able to search a repository for BPMN models/patterns for major activities, such as a letter of credit, customize the BPMN model by replacing patterns representing transactions or sub-transactions, with similar patterns for customization purposes to suit the specific context, and then use the TABS+R tool to transform the BPMN model into a smart contract and deploy it on the blockchain in automated fashion.

CONFLICT OF INTEREST

The authors declare no conflict of interest.

AUTHOR CONTRIBUTIONS

All three authors participated in research and writing of this paper.

REFERENCES

[1] D. Yang, C. Long, H. Xu, S. Peng, 2020. A Review on Scalability of Blockchain. In Proceedings of the 2020 The 2nd International Conference on Blockchain Technology(ICBCT'20). Association for Computing Machinery, New York, NY, USA, 1–6. DOI:https://doi.org/10.1145/3390566.3391665

[2] P. J. Taylor, T. Dargahi, Dehghantanha, R. M. Parizi, 2019. A Systematic Literature Review Of Blockchain Cyber Security - ScienceDirect. A systematic literature review of blockchain cyber security - ScienceDirect. https://www.sciencedirect.com/science/article/pii/S2352864818301536.

[3] S. Khan, F. Loukil, C. Ghedira-Guegan, E. Benkhelifa, A. Bani-Hani, 2021. Blockchain smart contracts: Applications, challenges, and future trends. Peer Peer Netw Appl. 2021 Apr 18:1-25. doi: 10.1007/s12083-021-01127-0. Epub ahead of print. PMID: 33897937; PMCID: PMC8053233.

[4] A. Vacca, A. Annalisa et al. 2021. A systematic literature review of blockchain and smart contract development: Techniques, tools, and open challenges. J. Syst. Softw. 174 (2021): 110891.

[5] R. Belchior, A. Vasconcelos, S. Guerreiro, M. Correia. 2021. A Survey on Blockchain Interoperability: Past, Present, and Future Trends. ACM Comput. Surv. 54, 8, Article 168, 41 pages. DOI:https://doi.org/10.1145/3471140

[6] K. Saito and H. Yamada, "What's So Different about Blockchain? — Blockchain is a Probabilistic State Machine," in Proc. 36th Int. Conf. on Distributed Computing Systems Workshops (ICDCSW), 2016, pp. 168–175. doi: 10.1109/ICDCSW.2016.28.

[7] J. A. Garcia-Garcia, N. Sánchez-Gómez, D. Lizcano, M. J. Escalona and T. Wojdyński, "Using Blockchain to Improve Collaborative Business Process Management: Systematic Literature Review," in IEEE Access, vol. 8, pp. 142312-142336, 2020, doi: 10.1109/ACCESS.2020.3013911.

[8] C. Lauster, P. Klinger, N. Schwab, F. Bodendorf, 2020. Literature Review Linking Blockchain and Business Process Management in 15th International Conference on Wirtschaftsinformatik, March 08-11, 2020, Potsdam, Germany. https://doi.org/10.30844/wi_2020_r10-klinger

[9] O. Levasseur, M. Iqbal, and R. Matulevičius, 2021. "Survey of Model-Driven Engineering Techniques for Blockchain-Based Applications". PoEM'21 Forum: 14th IFIP WG 8.1 Working Conference on the Practice of Enterprise Modelling.

[10] P. Tolmach, Y. Li, S. Lin, Y. Liu, Z. Li. 2021. A Survey of Smart Contract Formal Specification and Verification. ACM Comput. Surv. 54, 7, Article 148 (September 2022), 38 pages. DOI:https://doi.org/10.1145/3464421

[11] O. López-Pintado, L. García-Bañuelos, M. Dumas, I. Weber, and A. Ponomarev, "CATERPILLAR: A Business Process Execution Engine on the Ethereum Blockchain," Apr. 22, 2019, arXiv: arXiv:1808.03517. doi: 10.48550/arXiv.1808.03517.

[12] A. Tran, Q. Lu, and I. Weber, "Lorikeet: A Model-Driven Engineering Tool for Blockchain-Based Business Process Execution and Asset Management," in Proc. 2018 Int. Conf. on Business Process Management, 2018, pp. 1–5. [Online]. Available: https://api.semanticscholar.org/CorpusID:52195200

[13] J. Mendling et al., "Blockchains for Business Process Management—Challenges and Opportunities," ACM Transactions on Management Information Systems, vol. 9, no. 1, pp. 1–16, 2018, doi: 10.1145/3183367.

[14] F. Loukil, K. Boukadi, M. Abed, and C. Ghedira-Guegan, "Decentralized collaborative business process execution using blockchain," World Wide Web, vol. 24, no. 5, pp. 1645–1663, 2021.

[15] P. Bodorik, C. G. Liu, and D. B. Jutla, "TABS: Transforming automatically BPMN models into blockchain smart contracts," Elsevier Journal of Blockchain: Research and Applications, vol. 100115, pp. 1–26, 2023, doi: 10.1016/j.bcra.2022.100115.

[16] P. Bodorik, G. C. Liu, and D. Julta, "Using FSMs to Find Patterns for Off-Chain Computing," in Proc. 3rd Int. Conf. on Blockchain Technology, 2021, pp. 28–34. doi: 10.1145/3460537.3460565.

[17] C. Liu, P. Bodorik, and D. Jutla, "A Tool for Moving Blockchain Computations Off-Chain," in Proc. 3rd ACM Int. Symp. on Blockchain and Secure Critical Infrastructure, 2021, pp. 103–109. doi: 10.1145/3457337.3457848.

[18] C. Liu, P. Bodorik, and D. Jutla, "BPMN to smart contracts on blockchains: Transforming BPMN to DE-HSM multi-modal model," in Proc. 2021 Int. Conf. on Engineering and Emerging Technologies (ICEET), 2021, pp. 1–7. doi: 10.1109/ICEET53442.2021.9659771.

[19] C. Liu, P. Bodorik, and D. Jutla, "Long-Term Blockchain Transactions Spanning Multiplicity of Smart Contract Methods," in Proc. Int. Conf. on Blockchain and Trustworthy Systems (BlockSys'2023), Springer, 2023, pp. 141–155. doi: 10.1007/978-981-99-8104-5.

[20] C. G. Liu, P. Bodorik, and D. Jutla, "Automating Smart Contract Generation on Blockchains Using Multi-modal Modeling," Journal of Advances in Information Technology, vol. 13, no. 3, pp. 213–223, 2021, doi: 10.12720/jait.13.3.213-223.

[21] C. G. Liu, P. Bodorik, and D. Jutla, "Supporting Long-term Transactions in Smart Contracts," in Proc. Int. Conf. on Blockchain Computing and Applications (BCCA), 2022, pp. 11–19. doi: 10.1109/BCCA55292.2022.9922193.

[22] C. Liu, P. Bodorik, and D. Jutla, "Transforming Automatically BPMN Models to Smart Contracts with Nested Trade Transactions (TABS+)," Distrib. Ledger Technol., Apr. 2024, doi: 10.1145/3654802.

[23] Object–relational impedance mismatch. Accessed: Mar. 05, 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Object%E2%80%93relational_impedance_mismatch&oldid=1134321907.

[24] BPMN 2.0 Introduction - Flowable Open-Source Documentation. (n.d.). Retr. 2024/02/15 https://flowable.com/open-source/docs/.

[25] BPMN 2.0 Symbols—A complete guide with examples. (n.d.). Camunda. Retr. 2024/02/15 https://camunda.com/bpmn/reference/.

[26] Business Process Model and Notation (BPMN), Version 2.0.2. (n.d.). Retr. 2024/02/15 https://www.omg.org/spec/BPMN/2.0.2/PDF.

[27] About the Business Process Model and Notation Specification 2.0. (2010). Retr. 2024/02/15 https://www.omg.org/spec/bpmn/2.0/About-BPMN.

[28] Camunda (n.d.). Process Orchestration for end-to-end automation. Retr. 2024/02/15 https://camunda.com.

[29] Dikmans, L. (2008). Transforming BPMN into BPEL: Why and How. Oracle Middleware/Technical Details/Technical Article. Retr. 2024/10/16 https://www.oracle.com/technical-resources/articles/dikmans-bpm.html.

[30] Yannakakis, M. (2000). Hierarchical State Machines. Proc. IFIP Int. Conf. on Theoretical Computer Science, Exploring New Frontiers of Theoretical Informatics, pp. 315–330, ACM.

[31] A. Girault, B. Lee, and E. A. Lee, "Hierarchical finite state machines with multiple concurrency models," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 18, no. 6, pp. 742–760, 1999, doi: 10.1109/43.766725.

[32] C. A. R. Hoare, "Communicating sequential processes," Commun. ACM, vol. 21, no. 8, pp. 666–677, Aug. 1978, doi: 10.1145/359576.359585.

[33] Cassandras, C. (1993). Discrete event systems: Modeling and performance analysis. CRC Press. 1st ed. ISBN 10: 0256112126

[34] O. López-Pintado, M. Dumas, L. García-Bañuelos, and I. Weber, "Controlled flexibility in blockchain-based collaborative business processes," Information Systems, vol. 104, p. 101622, Feb. 2022, doi: 10.1016/j.is.2020.101622.

[35] C. Di Ciccio et al., "Blockchain Support for Collaborative Business Processes," Informatik Spektrum Journal, vol. 42, pp. 182–190, 2019.

[36] A. Bagozi, D. Bianchini, V. De Antonellis, M. Garda, and M. Melchiori, "A Three-Layered Approach for Designing Smart Contracts in Collaborative Processes," in On the Move to Meaningful Internet Systems: OTM 2019 Conferences, H. Panetto, C. Debruyne, M. Hepp, D. Lewis, C. A. Ardagna, and R. Meersman, Eds., Cham: Springer International Publishing, 2019, pp. 440–457. doi: 10.1007/978-3-030-33246-4_28.

[37] A. Mavridou and A. Laszka, "Designing Secure Ethereum Smart Contracts: A Finite State Machine Based Approach," in Proc. Int. Conf. Financial Cryptography and Data Security, Springer LNCS, 2018, pp. 523–540. doi: 10.48550/arXiv.1711.09327.

[38] A. Mavridou and A. Laszka, "Tool Demonstration: FSolidM for Designing Secure Ethereum Smart Contracts," in Proc. Principles of Security and Trust (POST 2018), Springer LNCS, 2018, pp. 217–277.

[39] A. Mavridou, A. Laszka, E. Stachtiari, and A. Dubey, "VeriSolid: Correct-by-Design Smart Contracts for Ethereum," Jan. 20, 2019, arXiv: arXiv:1901.01292. doi: 10.48550/arXiv.1901.01292.

[40] J. Jin, L. Yan, Y. Zou, J. Li, and Z. Yu, "Research on Smart Contract Verification and Generation Method Based on BPMN," Mathematics, vol. 12, p. 2158, Jul. 2024, doi: 10.3390/math12142158.

[41] M. Rodler, W. Li, G. O. Karame, and L. Davi, "{EVMPatch}: Timely and Automated Patching of Ethereum Smart Contracts," presented at the 30th USENIX Security Symposium (USENIX Security 21), 2021, pp. 1289–1306. Accessed: Aug. 07, 2024. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/rodler

[42] Z. Li, Y. Zhou, S. Guo, and B. Xiao, "SolSaviour: A Defending Framework for Deployed Defective Smart Contracts," in Proceedings of the 37th Annual Computer Security Applications Conference, in ACSAC '21. New York, NY, USA: Association for Computing Machinery, Dec. 2021, pp. 748–760. doi: 10.1145/3485832.3488015.

[43] H. Jin, Z. Wang, M. Wen, W. Dai, Y. Zhu, and D. Zou, "Aroc: An Automatic Repair Framework for On-Chain Smart Contracts," IEEE Transactions on Software Engineering, vol. 48, no. 11, pp. 4611–4629, Nov. 2022, doi: 10.1109/TSE.2021.3123170.

[44] F. Corradini, A. Marcelletti, A. Morichetta, A. Polini, B. Re, and F. Tiezzi, "Flexible Execution of Multi-Party Business Processes on Blockchain," in 2022 IEEE/ACM 5th International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), May 2022, pp. 25–32. doi: 10.1145/3528226.3528369.

[45] F. Corradini, A. Marcelletti, A. Morichetta, A. Polini, B. Re, and F. Tiezzi, "Engineering trustable choreography-based systems using blockchain," Mar. 2020, pp. 1470–1479. doi: 10.1145/3341105.3373988.

[46] G. Falazi, M. Hahn, U. Breitenbücher, and F. Leymann, "Modeling and execution of blockchain-aware business processes," SICS Softw.-Inensiv. Cyber-Phys. Syst., vol. 34, no. 2, pp. 105–116, Jun. 2019, doi: 10.1007/s00450-019-00399-5.

[47] G. Falazi, M. Hahn, U. Breitenbucher, F. Leymann, and V. Yussupov, "Process-Based Composition of Permissioned and Permissionless Blockchain Smart Contracts," 2019 IEEE 23rd International Enterprise Distributed Object Computing Conference (EDOC), pp. 77–87, Oct. 2019, doi: 10.1109/EDOC.2019.00019.

[48] P. Klinger, L. Nguyen, and F. Bodendorf, "Upgradeability Concept for Collaborative Blockchain-Based Business Process Execution Framework," in Blockchain – ICBC 2020, Z. Chen, L. Cui, B. Palanisamy, and L.-J. Zhang, Eds., Cham: Springer International Publishing, 2020, pp. 127–141. doi: 10.1007/978-3-030-59638-5_9.

[49] V. Buterin, "Ethereum White Paper: A Next Generation Smart Contract & Decentralized Application Platform." 2015. [Online]. Available: https://blockchainlab.com/pdf/Ethereum_white_paper-a_next_generation_smart_contract_and_decentralized_application_platform-vitalik-buterin.pdf