Blockchain: Research and Applications 6 (2025) 100285

Contents lists available at ScienceDirect

Blockchain: Research and Applications

journal homepage: www.journals.elsevier.com/blockchain-research-and-applications

Research Article

# Automated mechanism to support trade transactions in smart contracts with upgrade and repair

Christian Gang Liu [a,*], Peter Bodorik [a], Dawn Jutla [b]

[a] Faculty of Computer Science, Computer Science, Halifax, Canada
[b] Sobey School of Business, Saint Mary's University, Halifax, Canada

ARTICLE INFO

*Keywords:*
Blockchain
Smart contracts
BPMN
Transaction mechanism
Automated generation
Smart contract upgrade or repair
TABS+R

ABSTRACT

In our previous research, we addressed the problem of automated transformation of models, represented using the business process model and notation (BPMN) standard, into the methods of a smart contract. The transformation supports BPMN models that contain complex multi-step activities that are supported using our concept of multi-step nested trade transactions, wherein the transactional properties are enforced by a mechanism generated automatically by the transformation process from a BPMN model to a smart contract. In this paper, we present a methodology for repairing a smart contract that cannot be completed due to events that were not anticipated by the developer and thus prevent the completion of the smart contract. The repair process starts with the original BPMN model fragment causing the issue, providing the modeler with the innermost transaction fragment containing the failed activity. The modeler amends the BPMN pattern on the basis of the successful completion of 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 a new smart contract, ensuring consistent data and logic transitions. We previously developed a tool, called TABS+, as a proof of concept (PoC) to transform BPMN models into smart contracts for nested transactions. This paper describes the tool TABS+*R*, developed by extending the TABS+ tool, to allow the repair of smart contracts.

## 1. Introduction

The publication of the Bitcoin white paper in 2008 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, such as 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.

Researchers are 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 from 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 et al. [2], Khan et al. [3], Vacca et al. [4], Belchior et al. [5], Saito et al. [6], Garcia-Garcia et al. [7], Lauster et al. [8], and Levasseur et al. [9]. To simplify smart contract development, studies by López-Pintado et al. [11], Tran et al. [12], Mendling et al. [13], and Loukil et al. [14] propose to express the requirements of a blockchain application using a model expressed in the business process model and notation (BPMN), which 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 hierarchical state machine (DE-HSM) modeling to transform the BPMN model into a DE-HSM model that allows for graph-based representations of distributed blockchain applications, facilitating analysis and identifying patterns that remain isolated from other concurrent activities. We describe our

* Corresponding author.
*E-mail addresses:* Chris.Liu@dal.ca (C.G. Liu), Peter.Bodorik@dal.ca (P. Bodorik), Dawn.Jutla@gmail.com (D. Jutla).

https://doi.org/10.1016/j.bcra.2025.100285
Received 3 May 2024; Received in revised form 2 December 2024; Accepted 11 December 2024
Available online 28 March 2025
2096-7209/© 2025 The Authors. Published by Elsevier Ltd on behalf of Zhejiang University Press. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

approach in Refs. [15–22], together with a TABS tool (Tool to Automatically transform a BPMN model to Smart contract methods) that we developed as a proof of concept (PoC) to demonstrate the feasibility of our approach. In Ref. [22], we expand our approach presented in Ref. [15] to address collaborative activities in trade and distributed finance, to which we refer simply as trade activities. These activities are often performed by several participants executing various multi-step activities, such as price negotiations, letters of credit, transportation, and exchanges of various documentation. A smart contract naturally represents such collaborative activities employing several methods, and synchronization of such activities is thus required.

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 a 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 made by an execution of any of the methods of a smart contract (which of course may invoke other methods). The native blockchain transactions thus cannot include updates to the ledger made by two smart contract methods that were independently invoked by the distributed application. 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 multi-step trade transaction, simply referred to as 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 features to provide access control and privacy. 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 and identifies the methods forming the trade transaction, and the transformation process from BPMN to smart contracts uses our methodology to support the multi-step trade transaction properties. We also support nested trade transactions, while imposing some restrictions to ensure that trade transaction methods refer only to objects and methods within their defined scope. Since our initial proposal in Ref. [19], 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. However, 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 is multi-step and may be nested, involves the execution of multiple methods of a smart contract; thus, recovery from a failure is more complex than recovery from a failure of a native blockchain transaction. A recovery procedure needs to ensure that (i) the ledger is not affected by a failed transaction, and that (ii) different actors that participate in the trade transaction execution are informed of the failures in the correct sequence so that they can recover their resources dedicated to the execution of the failed trade transaction on their local systems.

In addition to invoking recovery procedures for the application, we also address the issue of the failure of smart contracts when real-life situations prevent their completion. In real life, if trade activity arrangements cannot be completed due to some events or conditions, alternative arrangements are made. However, if such trade activities are modeled by a smart contract, the question is how to update or repair the smart contract to represent the trade activity with new alternative arrangements.

The software application life cycle includes upgrades to fix bugs and introduce new features to respond to new requirements caused by the ever-changing environment. Smart contracts are not any different, but owing to the blockchain immutability, upgrading smart contracts causes difficulties, with active research addressing the problem, as judged by surveys on the topic [24–26]. However, real life complicates issues even further. A situation may arise in which a trade activity cannot be completed using the original arrangements due to some unanticipated events or conditions, and new arrangements need to be made. Thus, a smart contract developed to represent the original activities needs to be upgraded to represent the newly arranged activities to facilitate successful completion. Thus, not only does the smart contract need to be upgraded, but, if possible, the upgrade should avoid redoing completed activities, and we refer to such an upgrade as a smart contract repair. The question arises as to how to repair the activities of a smart contract while retaining the partially completed activities and ensuring consistency, which we also address in this paper.

### 1.1. Objectives and contributions

In our previous research [15–22], we addressed the issue of generating smart contracts from BPMN models with the support of nested transactions, defined over a subset of methods of a smart contract, to support multi-step trade transactions performed by several transaction participants. Developers declare a trade (sub)transaction as a collection of methods, while the automated transformation from a BPMN model to the methods of smart contracts also provides an automated transaction mechanism to support the multimethod and possibly nested transactions.

In this paper, we describe recovery procedures for the multi-step trade transactions and our approach to the smart contract repair. Our approach not only supports recovery from failure but also facilitates repair of a smart contract: if the execution of a trade activity, as represented by the smart contract, fails due to an unanticipated situation, alternative arrangements are made in order to complete the trade. Such alternative arrangements strive to reuse already completed trade activities in order to reduce the overall cost. However, the smart contract also needs to be upgraded to represent the new alternative arrangements that also avoid recovering and redoing activities that have been already completed successfully by the original smart contract before its failure exception occurred.

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 the state just before the transaction begins. This recovery includes not only the restoration of the transaction on the blockchain 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 PoC that demonstrates the feasibility of the proposed methodology for transaction repair.

To briefly summarize our contributions, 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. We describe a tool, called TABS+*R*, which we developed as a PoC.

### 1.2. Outline

Section 2 provides the necessary background. Section 3 details the recovery process for failed trade (sub)transactions, which restores the system to the state prior to 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 of trade activity failure, represented by trade transactions that require amendments, Section 4 describes our approach to repairing trade transactions on the basis of 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.

Section 5 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.

Section 6 provides an overview of related work in the field and discusses limitations. Finally, Section 7 presents the conclusions of the study and describes future research directions.

## 2. Background

This section first provides an overview of BPMN modeling and then discusses modeling with finite state machines (FSMs), hierarchical state machines (HSMs), and multi-modal modeling. We also review transactions in database and blockchain systems, comparing their properties with our concept of trade transactions. These concepts are foundational for our approach to repairing trade transactions generated from BPMN models. As this research extends our previous work on the automated generation of smart contracts from BPMN models, this section overviews our approach to automatically generating smart contracts from BPMN models.

### 2.1. Business process model and notation (BPMN)

BPMN, developed by the object management group (OMG) [27–30], 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 [31], while Oracle Corporation converts BPMN models into executable process blueprints via the business process execution language (BPEL) [32].

Key features of BPMN models include flow elements that represent computation flows between BPMN elements. A task represents a 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.

### 2.2. FSMs, HSMs, and multi-modal modeling

FSM modeling is widely used in software design and implementation and is often extended with features such as guards in FSM transitions. In the late 1980s, FSMs evolved into HSMs, incorporating hierarchical structures to facilitate pattern reuse, allowing states to contain other FSMs [33].

Girault et al. [34] described combining HSM modeling with concurrency semantics from models like communicating sequential processes [35] and discrete events [36]. They describe how a system state can be represented by an HSM, with a specific concurrency model, which 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.

### 2.3. BPMN model transformation to smart contract methods

In Refs. [15,22], we presented a methodology for transforming BPMN models into smart contracts. The transformation process involves several key steps:

1. **Transformation of the BPMN model to directed acyclic graph (DAG) representation:** The BPMN is pre-processed and is converted into a DAG representation. The mapping ensures that for any DAG vertex or edge, the corresponding BPMN element can be identified, and vice versa.
2. **Identification of single-entry single-exit (SESE) subgraphs:** The DAG is analyzed to identify SESE subgraphs. A SESE subgraph is such that it has a single-entry vertex, i.e., the only vertex in the subgraph that has an input edge from a vertex outside the subgraph, and a single-exit vertex, i.e., the only vertex in the subgraph that has an output edge leading to a vertex outside the subgraph. All other subgraph vertices have only edges connected to the internal nodes of that subgraph. SESE subgraphs are significant because they represent the flow of computation, and once the computation flow, represented by the graph edges, enters a SESE subgraph, it remains confined within that subgraph until it exits via the subgraph's exit node. This ensures a contained and manageable flow of computation. The identified SESE subgraphs are shown to the developer, who decides which of those subgraphs should be implemented by the transformation process as transactions. Any chosen SESE subgraph that contains other proper SESE subgraphs will be implemented as a parent transaction containing nested subtransactions, one for each of its SESE subgraphs, which is applied recursively.
3. **Transformation to discrete event-finite state machine (DE-FSM) model:** The DAG, with its identified SESE subgraphs, is transformed into a 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. The DE-HSM model is further detailed by elaborating each of the HSMs until the whole BPMN model is flattened into a network of DE-FSM sub-models.
4. **Transformation into the smart contract methods:** The interconnected DE-FSM models are then transformed into a 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.

A smart contract is essentially 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, which indicates that on a transition to that state, a particular task should be executed. The collaborative activities are thus represented by state changes in concurrent FSMs and are independent of the blockchain infrastructure. Thus, only the execution of the independently executed BPMN tasks is blockchain dependent.

We exploited the concept of multi-modal modeling and independent subgraphs in Ref. [15], and then in our subsequent work on the project, we support sidechain processing by enabling the developer to choose and 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 the sidechain. If computation performed on a sidechain is much cheaper than on the mainchain, then sidechain processing may be beneficial.

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

The transaction mechanism, to support the nested multi-step trade transactions, is generated by the transformation process from a BPMN model to a smart contract using a pattern augmentation scheme [22]. Ledger writes are not applied to the ledger directly, but instead, are cached and then applied to the ledger only during the trade transaction commit phase after all ledger updates have been cached. Therefore, if a blockchain transaction fails, ledger recovery is unnecessary. However, participants must be informed of the failure so that they can release local resources allocated for the failed transaction's processing.

## 3. Recovery procedures for nested trade transactions

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

### 3.1. 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, thus complicating the recovery process.

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

1. The ledger state remains unaffected by the failed transaction's execution.
2. All participants are notified of the failure so that 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 subtransactions must be executed before those of the parent transaction.

### 3.2. Trade activities, nested transactions, and recovery procedures

Consider a simple example of a smart contract that supports a trade transaction for the 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 the 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. Fig. 1 shows the trade activities as a BPMN model created using the Camunda platform invoked from our TABS+*R* tool, which 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. Fig. 1 represents the following trade activities:

- PriceAndEscrow includes price negotiation and escrow payment.
- GetTrRequirements includes determining the transport safety requirements.
- GetRailInsurance includes obtaining the rail insurance to cover the product transport while satisfying the safety requirements.
- GetRailTransporter includes hiring the company to transport the product.
- DoTransport includes the transport of the product.
- ReceiveAndFinalize includes the customer's acceptance of the delivered product and completion of the payment.

Recall that the transformation process, from a BPMN model into the methods of a smart contract, uses the concept of SESE subgraphs to find BPMN patterns that are suitable to be treated as transactions. Assume that the business analyst chooses the nested transactions shown in Fig. 2. The figure was generated from Fig. 1 by hand-drawing dashed-line rectangles over Fig. 1 to represent the nested transactions. Thus, full-line rectangles in Fig. 2 represent the trade activities, which were already shown in Fig. 1, while the dashed-line rectangles represent (sub)transactions that have names ending with the string "*_tx*". Fig. 2 thus shows the following nested transactions:

- *priceAndEscrow_tx* includes the PriceAndEscrow activity.
- *transportProduct_tx* includes subtransactions *getTrRequirements_tx* (determining transport requirements, obtaining rail insurance, and obtaining a transporter) and *doTransport_tx* (actual product transport).
  - *getTrRequirements_tx* subtransaction includes the GetTrRequirements, GetRailInsurance, and GetRailTransport activities.
  - *doTransport_tx* subtransaction includes the DoTransport activity.
- *receiveAndFinalize_tx* includes the ReceiveAndFinalize trade activity to finalize the transaction by receiving the product and completing the 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 subtransactions. Each recovery

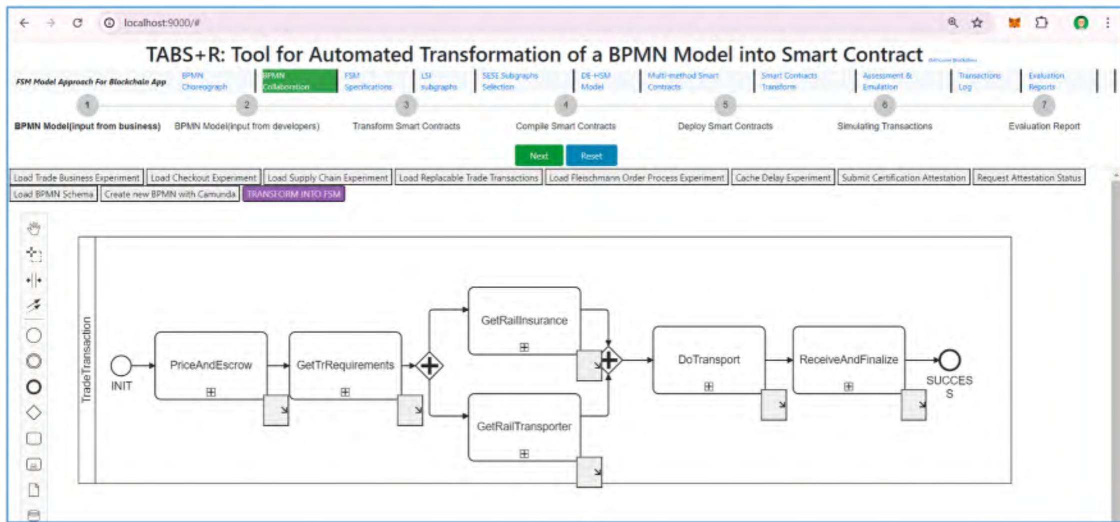


**Fig. 1.** Block diagram of trade activities represented using a business process model and notation (BPMN) model created using Camunda platform [31].

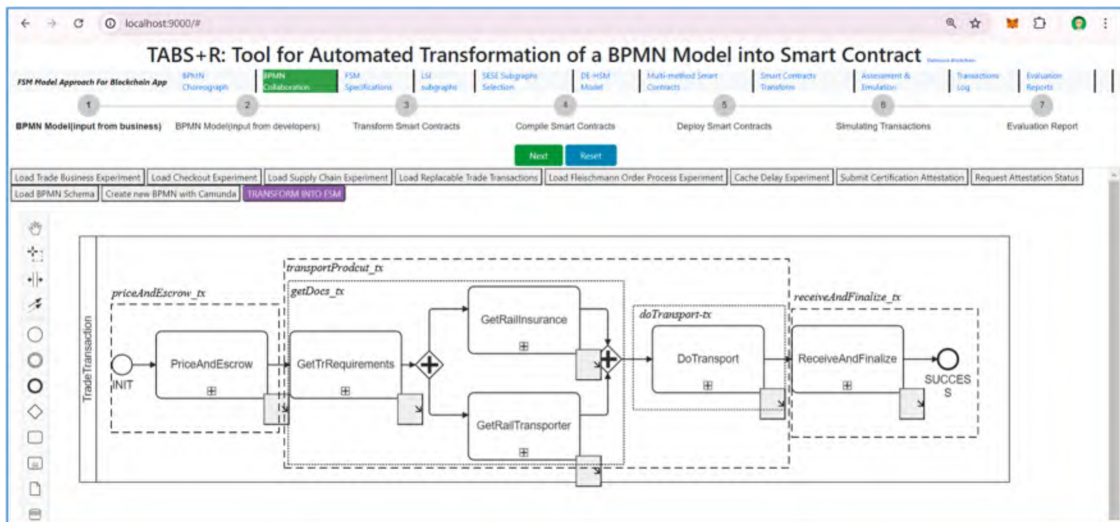


**Fig. 2.** BPMN model of trade activities as nested trade transactions.

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_tx*, wherein the trade transaction *transportProduct_tx* includes subtransactions *getDocs_tx* and *doTransport_tx*, as shown in Fig. 2.

In addition to the transactions shown in Fig. 2, 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 subtransaction are invoked from its parent transaction, while the methods of a trade transaction that is not a subtransaction are invoked from the main contract that contains all non-trade-transaction methods. In case of failure, recovery procedures for trade subtransactions 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.

## 4. Trade transaction upgrade and repair

Developers strive to anticipate potential issues that may arise during the execution of trade transactions and write exception handlers to manage them, trying to ensure the 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, when an unanticipated and uncaught exception occurs, the question arises about how to complete the trade activities when a part of the smart contract fails. Given the immutability of blockchains, representing new arrangements in the smart contracts is challenging. One approach could involve creating a new smart contract tied to the failed one while successfully leveraging 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 successfully complete the trade activity.

When a trade activity represented by a smart contract fails, the main objective is to make alternative arrangements that will overcome the failure, and then amend or repair the smart contract with the alternative arrangements 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 to repairing trade (sub)transactions in smart contracts to reflect alternative arrangements utilizes nested trade transactions to facilitate repair, focusing on amending the failed subtransaction rather than the entire trade activity. We automate the generation and deployment of smart contracts and describe how to update the failed subtransactions to ensure continued execution of the smart contract post-repair. 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 to complete the repair within the identified BPMN pattern that corresponds to the trade transaction, then the repair of that pattern is aborted and 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 the generation of the smart contract for the repaired pattern and how we achieve replacement, i.e., an upgrade of the failed smart contract with the repaired version.

### 4.1. Repair at the BPMN model level

The developer is presented with the original BPMN pattern that led to the failure, including the failure's cause. Then, the developer must replace this failure pattern with a new one that presumably avoids the failure. The repaired BPMN model integrates the 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 Fig. 3.

The initial step in Fig. 3 involves determining which BPMN patterns within the model caused the failure. The cause of the 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 context of the BPMN model is crucial. The developer uses this information to amend the BPMN pattern and repair the trade activity.

First, the failure information must be translated or 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 or repaired and the reasons for failure. The developer 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 replaces the BPMN pattern causing the failure with a repaired BPMN pattern, which is then transformed into the methods of a smart contract that replaces its failed version.

Consider a scenario where *doTransport_tx* fails due to a washed-out rail line. The BPMN model shows the developer 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/or transporter, the repair must "backtrack" to the parent transaction that also includes the GetRailInsurance and GetRailTransporter activities. If the repaired pattern does not meet the required outputs for subsequent activities, the repair extends to these activities, as outlined in Fig. 3. 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 carry out the actual trade activities.

Continuing with our example, assume that an alternative arrangement is found for the product transport, with the same transporter using an alternative rail-line route and existing insurance applied 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 and GetTransporter sub-activities do not need to be repaired. This is the case when the pre-repair condition (a) of step 2 in the repair outline shown in Fig. 3 is satisfied.

However, consider a situation where *doTransport_tx* fails, and no other rail route is available, necessitating road transport. Assuming that the hired rail transporter does not provide road transport, or the insurance for the rail transport does not apply to the road transport, then the pre-repair condition (a) in step 2 is not met. Hence, 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 innermost trade subtransaction is aborted, and the repair of the previously completed activities must be amended to accommodate alternative transport arrangements. In such a situation, the repair

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 and information involved. The goal is to replace *Pf* with a repaired pattern (*Pr*) under the following constraints:
   (a) Pre-repair condition: The computation in the new pattern uses the same objects that were input to the failed execution.
   (b) Post-repair condition: 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 the above 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.

**Fig. 3.** Repair steps.

escalates to the parent transaction. Thus, the repair restarts for the BPMN pattern representing the parent trade transaction, *transportProduct_tx*. Since the *transportProduct_tx* parent transaction includes the GetInsurance and GetTransporter sub-activities and the failed *transportProduct_tx* subtransactions, the developer must amend the BPMN pattern, including the GetInsurance, GetTransporter, and DoTransport trade activities.

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. However, if the repaired pattern does not provide all information required for the subsequent activities, repair extends to the parent trade (sub)transaction. This represents the situation where the post-repair condition (b) of step 2 in Fig. 3 is not satisfied. The final step uses the amended 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 PoC is described in the next section.

### 4.2. Repair: from BPMN model to smart contract

In this subsection, we detail each repair process step shown in Fig. 3, focusing on transformations between the BPMN model and smart contract abstraction levels.

#### 4.2.1. 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.

Additionally, recall that failure is detected during the original smart contract's execution when an unhandled exception occurs. 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 that is 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.

#### 4.2.2. Model amendment

After presenting the BPMN model and failure information to the developer, the developer attempts to repair the identified, failed BPMN pattern to generate a new, repaired BPMN pattern. The repair must satisfy the following constraints:

(a) Pre-repair condition: 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) Post-repair condition: 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. Thus, it must be ensured that information required for executing the pattern being repaired is produced by previously completed activities as per the pre-repair constraint. In our example, if the same transporter and insurance can be used for the alternative 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.

The post-repair constraint ensures that the repaired pattern's computation produces the information required for subsequent computations. This means that the objects and information flowing out of the repaired BPMN pattern *Pr* must include those flowing out of the failed pattern *Pf*.

#### 4.2.3. Repaired smart contract generation

Each trade (sub)transaction is packaged and deployed in a separate smart contract. To facilitate replacing the 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 the 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. Applications do not directly invoke smart contract methods; instead, they invoke an application programming interface (API) prepared by the TABS+*R* tool, which in turn invokes the smart contract methods. To "upgrade/repair" the trade (sub)transaction such as *transportProduct_tx*, the API is updated to point to the new smart contract version and its methods, ensuring the invocation of the new *transportProduct_tx* transaction version that replaces the failed version.

### 4.3. Discussion

Before repairing a BPMN pattern, alternative arrangements must be found by a business analyst or the developer, a process outside this paper's scope. However, once such arrangements are made, their representation in the BPMN pattern under repair is supported by our repair system, providing the developer with information on the objects flowing into and out of the computation performed by a trade (sub)transaction. Although we are progressing toward supporting smart contract generation for trade transactions and creating infrastructure for the automated generation and repair of trade transactions in the trade of goods and services, there are still aspects of utilizing the concept of nested transactions that need further investigation, which we will discuss below.

#### 4.3.1. Evaluating the pre- and post-repair conditions

Although the developer has information on the objects and information flowing into the BPMN fragment to be repaired, currently, the decision whether the repaired fragment satisfies the pre- and post-repair conditions is left to be made by the developer. Clearly, further assistance should be provided to the developer. For instance, insurance is required for the rail transport. If the rail line is washed out and the road transport is needed instead, how can it be recognized in an automated fashion that the insurance for the rail does not apply to the road transport? Currently, we do not assist the developer in making such decisions.

#### 4.3.2. Selection of nested transactions and their overhead

Nested trade transactions incur high overhead because an atomic

commitment of subtransactions, within a parent transaction, is coordinated via the 2-Phase Commit (2 PC) protocol, which requires the order of $n^2$ messages for $n$ participants. Consequently, the decision by the developer to deploy BPMN SESE subgraphs as nested subtransactions in the smart contract should not be made lightly. For instance, if the activities of a SESE subgraph manipulate only digital assets in such a way that a subtransaction activities can be easily compensated, then the use of subtransactions may be avoided by the developer carefully orchestrating compensating activities in the case of exceptions, which of course incurs the one-time overhead cost due to the developer's time to orchestrate the compensating activities. However, if the one-time task of writing code for the compensating activities by developers is high due to their complexity, automated generation of recovery procedures when nested subtransactions are used may be beneficial if the cost of the 2 PC protocol for nested transactions is less than that of the developer time to write the compensating activities.

Another consideration is the cost associated with aborting an activity/task that has been partially completed and is being recovered . Consider, for instance, a simple case of ordering a number of parts that are to be assembled into a physical product under a deadline. First, parts are ordered, and when they arrive, the assembly and delivery of the product proceed. The issue is that if the ordering of any part fails, then the product assembly fails due to missing the deadline. Consider, for example, the following two scenarios:

i. An order for a part may be canceled without any penalty.
ii. Once an order for a part is placed, canceling it incurs a penalty as the supplier initiates shipment via its sub-contractors immediately after an order is placed.

For (i) above, subtransactions are not required as placed orders for any parts can be canceled without any penalty through compensating activities represented by code produced by the developer. However, for the case (ii), the subtransactions are useful, as using them facilitates placing the orders for all parts only after it is ascertained that all parts are available to order, and hence avoiding the high cost of canceling orders.

Thus, using nested transactions is cost-effective if the cost of 2 PC protocol execution within a smart contract is less than the cost of canceling an order due to suppliers' penalties plus the cost of the developer time to orchestrate compensating activities if canceling already placed orders for parts. Clearly, further research on when to use nested transactions is needed.

## 5. 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 while also providing a mechanism to support the transactional properties. To evaluate the feasibility of smart contract repair, we further augmented TABS+ into TABS+*R*, which 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 supports issuing multiple transactions and measuring various execution 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 Figs. 1 and 2 as a case study. Next, we explain how the tool assists developers in facilitating repair and resuming trade activities.

### 5.1. *Overview of the TABS+R tool*

TABS+*R* utilizes the Camunda platform [31] for creating BPMN models according to BPMN specifications [27–30]. These models are stored in XML format. Fig. 1 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 and providing code for template methods that represent the BPMN task elements. The tool supports the generation of smart contracts for Hyperledger Fabric blockchains or Ethereum virtual machine (EVM)-based blockchains, such as Quorum or Ethereum. The mainchain smart contract can invoke methods of smart contracts deployed on a sidechain.

Fig. 4 shows a screenshot of the BPMN model transformed into the DE-HSM model with SESE subgraphs derived from the BPMN model of trade activities in Fig. 1. The left-hand panel displays the SESE subgraphs identified for the BPMN model by TABS+*R*. The right-hand panel lists these SESE subgraphs as selectable boxes, and the developer can instruct the system to deploy the selected subgraphs as trade (sub) transactions.

The developer decides which independent subgraph patterns should be deployed 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 the Ethereum-compatible sidechain, Quorum is used. After the selections, the model is transformed into methods of smart contracts that are then deployed and executed.

The developer can step through the system's execution, observing message-by-message progress. The tool graphically represents state changes in individual FSMs of the DE-FSM submodels (Fig. 5) to aid in testing and manual verification. Execution delays are displayed as the process proceeds. Additionally, a developer can generate exceptions for specific activities, which is useful for testing and evaluating transaction repair. In Fig. 5, fault exception propagation is highlighted in red.

### 5.2. *Repair of trade (sub)transactions 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 fragment corresponding to the innermost subtransaction containing the failed trade activity. The developer then amends the BPMN model to create a repaired version that resolves the failure (Fig. 6).

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 subtransaction and (b) the output information flowing out remain consistent with the previous failed version, i.e., the pre- and post-repair conditions are satisfied, and if not, the repair escalates to the parent transaction. Once the repair is achieved, a new smart contract for the repaired (sub)transaction is generated and deployed on the same blockchain as the original failed version. Additionally, the distributed application's API must be updated to invoke the repaired subtransaction instead of the failed one. Once these changes are made, the developer instructs the tool to proceed with the execution of the repaired subtransaction.

The achieved repair is contingent upon the applicability of input

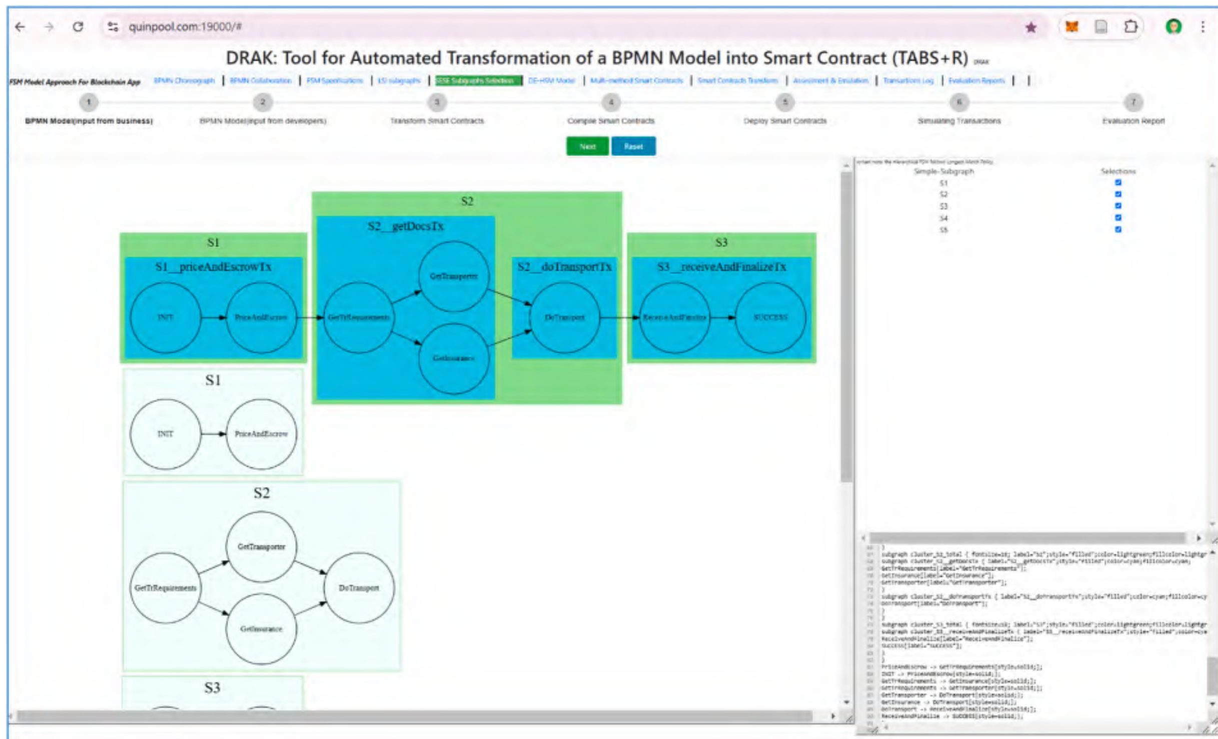


**Fig. 4.** Identified single-entry single-exit (SESE) subgraphs and their selection.

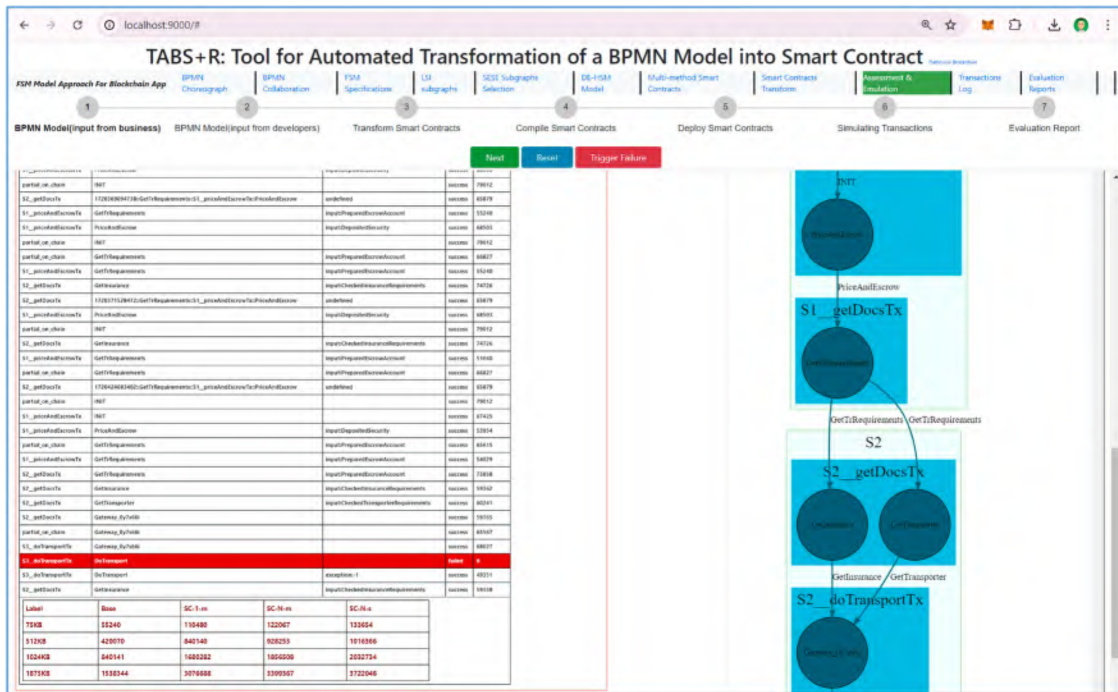


**Fig. 5.** Monitoring the execution of trade (sub)transactions and their activities for evaluation purposes.

information, such as insurance documents and transporter selection, to the new transport method. For our example, if the same transporter is able to provide road transport instead of rail transport, and the original insurance also covers road transport, then repair of the *doTransport_tx* subtransaction proceeds as planned with the same insurance contract and the same transporter contract as for the original smart contract. However, if either of the two conditions is not met, the repair of the subtransaction is aborted. In such a case, the developer is led to repair the parent transaction instead, as shown in Fig. 7. This process involves arranging new insurance and transport by road, which need to be recorded in the smart contract. The whole BPMN model after repair in Fig. 8 shows the new road-transport requirements and new arrangements for insurance and transport by road.

After completing the BPMN model for the repair, the developer

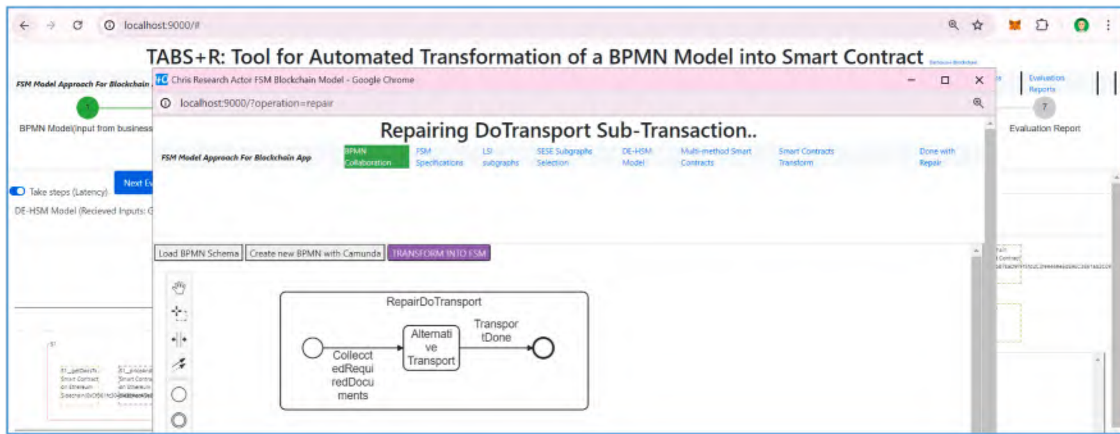


**Fig. 6.** Repair of the trade subtransaction *doTransport_tx*.

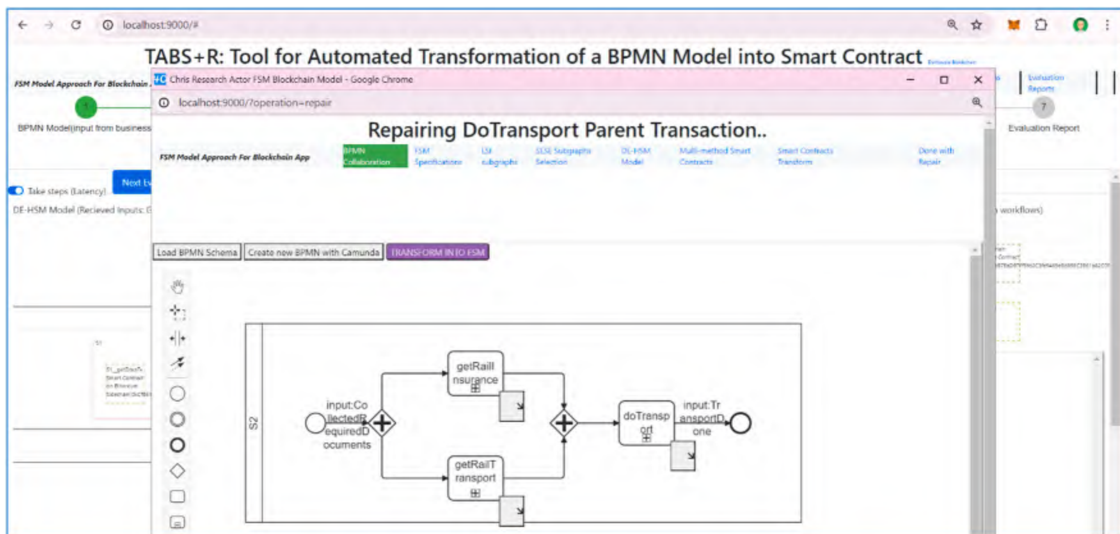


**Fig. 7.** Repair of the trade transaction *transportProduct_tx*, parent transaction of the failed *doTransport_tx*.

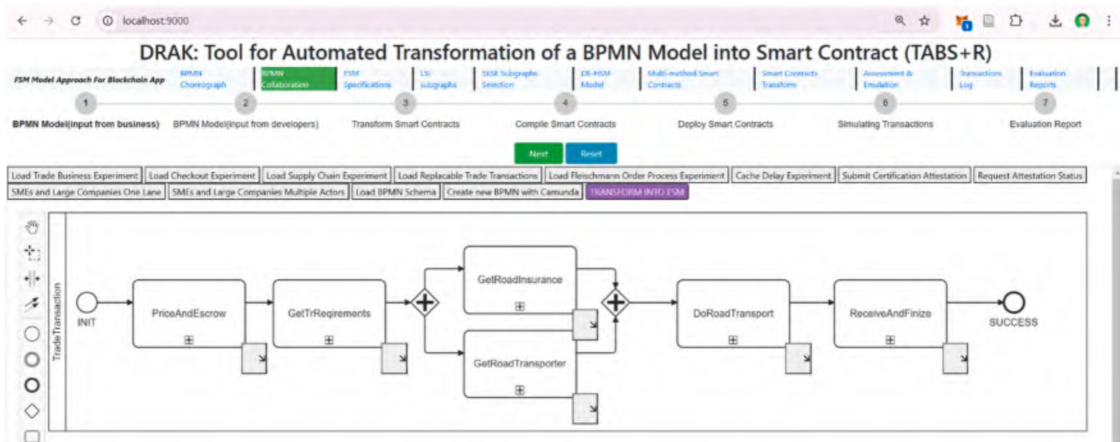


**Fig. 8.** BPMN diagram after the repair of the trade transaction *transportProduct_tx*, parent transaction of the failed *doTransport_tx*.

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 subtransaction and continue the overall workflow execution.

### *5.3. 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.

## 6. Related work, limitations, and future work

### 6.1. Related work

The Lorikeet project [12] uses a 2-phase approach to transform 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 distributed application. 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 nonfungible assets. It provides a registry of tokens and methods for asset management, such as transfers, thus enabling rapid prototyping of smart contracts from BPMN models that require such features. This allows for the creation, testing, and modification of smart contracts before deployment.

Caterpillar [11,37] 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. [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. [38] 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 Ref. [22], we expanded the comparison performed by Di Ciccio et al. [38] by including CoBuP [14] and TABS+ approaches and adding the following features for comparison purposes: support for nested transactions, deployment capabilities, type of synchronization, and privacy. The features that distinguish our approach and the TABS+ tool from others include the transformation of a BPMN into a DAG representation and then to the DE-HSM and DE-FSM models, which enable the analysis of process flow to identify localized processing using SESE subgraphs. Unlike other systems that use direct transformation of BPMN models into smart contract code, TABS+ produces smart contracts that are abstract representations of process flows with details 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, wherein each task has well-defined inputs and outputs. Furthermore, localized SESE graphs may be packaged and deployed as separate smart contracts, which may be deployed on sidechains for efficiencyor be used to define nested trade transactions.

Similar to CoBuP, Bagozi et al. [39] used a three-layer but simpler approach. In the first layer, the business analyst creates the BPMN model. 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, smart contracts are created and deployed on a specific blockchain.

Mavridou and Lazska [40,41] advocated the correct design of smart contracts. They target systems whose requirements are represented using an FSM that is transformed into the methods of a smart contract. Transformation may result in the 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 as a PoC, called FSolidM, that examines each smart contract method and augments it with security codes 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 et al. [42] advanced the work on FSolidM by creating a framework, called VeriSolid. The smart contract is specified via 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 or constraints. The transition system is analyzed using the declarations and guards, and is 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 verifiable properties.

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 the representation of the desirable properties in the BPMN model.

The verification of smart contracts generated from BPMN has been addressed in Ref. [43]. They use a two-step transformation process. In the first phase, the BPMN model is transformed to Prolog, which is used to validate the business logic. Following this, they transform the BPMN model into a smart contract in the GO language. However, there is no validation of the generated code.

When smart contracts are used to represent the collaborative activities of participants, it has been acknowledged that the blockchains' immutability property is desirable for promoting trust, but it also causes difficulties, as frequently, such collaborations need to be augmented to either repair security issues or support new desirable features. Thus, the upgradability of smart contracts is an important issue that has been tackled in literature, particularly in the context of security of smart contracts, as is demonstrated by literature surveys that have been performed on this topic [24–26]. For instance, Rodler et al. [44] described a framework, EVMPatch, which 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 technique to patch common bugs, such as integer overflows or underflows and access control errors.

Another bug fixing framework is SolSaviour [45] that uses a voteDestruct mechanism to allow contract stakeholders 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 into the new ones. Our tool TABS+*R* is similar when the repaired version of the smart contract incorporates the successfully completed activities of its unrepaired version.

Aroc [46] 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. Instead of using a proxy contract that invokes a patched or repaired and redeployed version of the vulnerable

smart contract, the authors propose a new smart contract that is created for preventing 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 redirection of the invocations of the vulnerable smart contract to the patch smart contract.

Corradini et al. [47,48] addressed the tension between the trust in blockchain, achieved via immutability, and the need for flexibility, which is required for multiparty collaboration. They used 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 the feasibility of their approach.

Falazi et al. [49] addressed the issue of using smart contracts in support of business process modeling and developed a prototype, BlockME, to validate their approach. They use BPMN to represent the choreography of the business processes while extending BPMN tasks to support invocation of smart contract methods for permissionless blockchains. The BPMN model is transformed into BPEL for execution. The choreography of processes is executed via BPEL that invokes smart contract methods that implement the BPMN task elements. Authors extend their approach and BlockME prototype in Ref. [50] 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 as opposed to the smart contract methods used in our TABS+*R* approach.

The closest work to our approach in upgrading smart contracts is Ref. [51]. This 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. The case they use 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. The upgrade thus needs to be carefully managed to ensure that activities already executed are consistent in the context of the upgrade. Their findings suggest that the unstructured storage proxy pattern is the most promising for practical use, especially regarding cost-effectiveness and minimal added complexity.

In comparison to Ref. [51], 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 newly upgraded smart contract.

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

### 6.2. Limitations and future work

Although we feel that our approach to easing 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. These are besides the issues of the high cost of nested transactions and the determination of the pre- and post-repair conditions that were discussed in Section 4.3.

#### 6.2.1. Securing smart contract methods

To secure smart contracts, we adopted the approach in Refs. [40,41], wherein the authors propose the hardening of smart contracts created by the transformation of an FSM to smart contract methods. Given that an FSM is 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) re-entrancy issues, (ii) transaction ordering in the face of unpredictable states, (iii) timed transitions, and (iv) access control. We successfully incorporated the re-entrancy protection into TABS+ by inserting appropriate locking patterns into the smart contract and supporting access control. In essence, the security patterns are inserted into the smart contract method at the start of a method and 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 the man-in-the-middle attack.

#### 6.2.2. Validation and verification

Validation and verification need to be an integral part of the transformation process from BPMN models to smart contract deployment. Transformation of a BPMN model results in a DE-FSM model that is a transition system, and we plan to apply the VeriSolid [42] 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 also extend that documentation with the desirable properties and then perform formal verification of the system in terms of liveliness, reachability, and deadlock-free properties, so that the desirable properties are satisfied.

#### 6.2.3. Blockchain agnostic smart contracts

One of our objectives is to achieve the generation of smart contracts that are blockchain agnostic. We made progress toward this objective by representing the collaboration in a blockchain independent manner by expressing it in terms of the interconnection of the DE-FSM models. However, currently, to apply a smart contract developed for one blockchain to be deployable and executable on another blockchain, the scripts for methods representing 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 Ref. [52], 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.

## 7. Summary and conclusions

The 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. Activities in the trade of goods and services are subject to effects from external events that may cause the 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 upgrade of a smart contract as a transaction repair, although others may use the terms upgrade or replacement. We described how we use the concept of nested trade transactions, together with the automatically generated transaction mechanism, to support the repair or 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 failed smart contract version may be incorporated consistently into the new version of the

smart contract.

However, when a trade activity needs to be repaired to respond to external environment changes, such repairs are performed by business analysts who can then repair the BPMN model representing the activity. 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's 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 amendment to the smart contract, and deploy the smart contract.

As can be appreciated, delays associated with allocating a developer's time and with the developer's time needed to write the new version of the smart contract replacing the failed transactions, cause difficulties especially in situations when the repair of a smart contract representing trade activities needs to be made promptly, such as 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 the assistance of a software developer. The major obstacle is the generation of the code for the methods that implement the PBMN task elements, currently performed by the developer. We are investigating the use of DMN modeling to represent the business logic of the methods implementing the BPMN task elements and transforming the DMN models automatically into the methods of a smart contract.

By supporting the automated creation of smart contracts from BPMN models and providing support for the augmentation of BPMN models with BPMN patterns and the 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 for major activities, such as a letter of credit, and customize the BPMN model by replacing patterns representing transactions or subtransactions, 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.

## CRediT authorship contribution statement

**Christian Gang Liu:** Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Project administration, Methodology, Investigation, Funding acquisition, Formal analysis, Data curation, Conceptualization. **Peter Bodorik:** Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Project administration, Methodology, Investigation, Funding acquisition, Formal analysis, Data curation, Conceptualization. **Dawn Jutla:** Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Project administration, Methodology, Investigation, Funding acquisition, Formal analysis, Data curation, Conceptualization.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## References


[1] D. Yang, C. Long, H. Xu, et al., A review on scalability of blockchain, in: Proceedings of the 2020 2nd International Conference on Blockchain Technology, ACM, 2020, pp. 1–6. https://doi.org/10.1145/3390566.3391665.

[2] P.J. Taylor, T. Dargahi, A. Dehghantanha, et al., A systematic literature review of blockchain cyber security, Digit. Commun. Netw. 6 (2) (2020) 147–156, https://doi.org/10.1016/j.dcan.2019.01.005.

[3] S.N. Khan, F. Loukil, C. Ghedira-Guegan, et al., Blockchain smart contracts: applications, challenges, and future trends, Peer Peer Netw. Appl. 14 (5) (2021) 2901–2925, https://doi.org/10.1007/s12083-021-01127-0.

[4] A. Vacca, A. Di Sorbo, C.A. Visaggio, et al., A systematic literature review of blockchain and smart contract development: techniques, tools, and open challenges, J. Syst. Softw. 174 (2021) 110891, https://doi.org/10.1016/j.jss.2020.110891.

[5] R. Belchior, A. Vasconcelos, S. Guerreiro, et al., A survey on blockchain interoperability: past, present, and future trends, ACM Comput. Surv. 54 (8) (2021) 1–41, https://doi.org/10.1145/3471140.

[6] K. Saito, H. Yamada, What's so different about blockchain?: blockchain is a probabilistic state machine, in: Proceedings of the 2016 IEEE 36th International Conference on Distributed Computing Systems Workshops (ICDCSW), IEEE, 2016, pp. 168–175. https://doi.org/10.1109/ICDCSW.2016.28.

[7] J.A. Garcia-Garcia, N. Sánchez-Gómez, D. Lizcano, et al., Using blockchain to improve collaborative business process management: systematic literature review, IEEE Access 8 (2020) 142312–142336, https://doi.org/10.1109/ACCESS.2020.3013911.

[8] C. Lauster, P. Klinger, N. Schwab, et al., Literature review linking blockchain and business process masnagement, in: Proceedings of the 15th International Conference on Wirtschaftsinformatik, WI 2020 Zent. Tracks, 2020, pp. 1802–1817. https://doi.org/10.30844/wi_2020_r10-klinger.

[9] O. Levasseur, M. Iqbal, R. Matulevičius, Survey of model-driven engineering techniques for blockchain-based applications, in: Proceedings of the 14th IFIP WG 8.1 Working Conference on the Practice of Enterprise Modelling, CEUR, 2021, pp. 11–20. https://ceur-ws.org/Vol-3045/paper02.pdf.

[10] P. Tolmach, Y. Li, S.W. Lin, et al., A survey of smart contract formal specification and verification, ACM Comput. Surv. 54 (7) (2021) 1–38, https://doi.org/10.1145/3464421.

[11] O. López-Pintado, L. García-Bañuelos, M. Dumas, et al., Caterpillar: a business process execution engine on the Ethereum blockchain, Softw. Pract. Exp. 49 (2018) 1162–1193, https://doi.org/10.1002/spe.2702.

[12] A. Tran, Q. Lu, I. Weber, Lorikeet: a model-driven engineering tool for blockchain-based business process execution and asset management, in: Proceedings of International Conference on Business Process Management, CEUR, 2018, pp. 1–5. https://api.semanticscholar.org/CorpusID:52195200.

[13] J. Mendling, I. Weber, W. Van Der Aalst, et al., Blockchains for business process management - challenges and opportunities, ACM Trans. Manage. Inf. Syst. 9 (1) (2018) 1–16, https://doi.org/10.1145/3183367.

[14] F. Loukil, K. Boukadi, M. Abed, et al., Decentralized collaborative business process execution using blockchain, World Wide Web 24 (5) (2021) 1645–1663, https://doi.org/10.1007/s11280-021-00901-7.

[15] P. Bodorik, C.G. Liu, D. Jutla, TABS: transforming automatically BPMN models into blockchain smart contracts, inBlockchain 4 (1) (2023) 100115, https://doi.org/10.1016/j.bcra.2022.100115. March.

[16] P. Bodorik, C.G. Liu, D. Julta, Using FSMs to find patterns for off-chain computing: finding patterns for off-chain computing with FSMs, in: Proceedings of the 2021 The 3rd International Conference on Blockchain Technology, ACM, 2021, pp. 28–34. https://doi.org/10.1145/3460537.3460565.

[17] C. Liu, P. Bodorik, D. Jutla, A tool for moving blockchain computations off-chain, in: Proceedings of the 3rd ACM International Symposium on Blockchain and Secure Critical Infrastructure, ACM, 2021, pp. 103–109. https://doi.org/10.1145/3457337.3457848.

[18] C. Liu, P. Bodorik, D. Jutla, From BPMN to smart contracts on blockchains: transforming BPMN to DE-HSM multi-modal model, in: Proceedings of the 2021 International Conference on Engineering and Emerging Technologies (ICEET), IEEE, 2021, pp. 1–7. https://doi.org/10.1109/ICEET53442.2021.9659771.

[19] C.G. Liu, P. Bodorik, D. Jutla, Long-term blockchain transactions spanning multiplicity of smart contract methods, in: J. Chen, B. Wen, T. Chen (Eds.), Blockchain and Trustworthy Systems, Springer, Singapore, 2024, pp. 142–155. https://doi.org/10.1007/978-981-99-8104-5_11.

[20] C.G. Liu, P. Bodorik, D. Jutla, Automating smart contract generation on blockchains using multi-modal modeling, J. Adv. Inf. Technol. 13 (3) (2022) 213–223, https://doi.org/10.12720/jait.13.3.213-223.

[21] C.G. Liu, P. Bodorik, D. Jutla, Supporting long-term transactions in smart contracts, in: Proceedings of the 2022 Fourth International Conference on Blockchain Computing and Applications (BCCA), IEEE, 2022, pp. 11–19. https://doi.org/10.1109/BCCA55292.2022.9922193.

[22] C. Liu, P. Bodorik, D. Jutla, Transforming automatically BPMN models to smart contracts with nested trade transactions (TABS+), ACM J., Distributed Ledger Technol., Volume 3, Issue 3, Article No.: 21, Pages 1–37, doi:10.1145/3654802.

[23] Object–relational impedance mismatch. https://en.wikipedia.org/w/index.php?title=Object%E2%80%93relational_impedance_mismatch&oldid=1134321907. (Accessed: 5 Mar 2023).

[24] S. Meisami, W.E. Bodell III, A comprehensive survey of upgradeable smart contract patterns, arXiv (2023) preprint.

[25] N.M. Girish, S. Kaganurmath, Upgradability of smart contracts: a review, Intern. Res. J. Eng. Technol. 9 (7) (2022) 2603–2606.

[26] S. Palladino, The state of smart contract upgrades. https://blog.openzeppelin.com/the-state-of-smart-contract-upgrades, 2020. (Accessed: 6 Oct 2020).

[27] BPMN 2.0 introduction—Flowable open-source documentation. https://flowable.com/open-source/docs/. (Accessed: 15 Feb 2024).

[28] BPMN 2.0 symbols—A complete guide with examples. https://camunda.com/bpmn/reference/. (Accessed: 15 Feb 2024).

[29] Business process model and notation (BPMN), version 2.0.2. https://www.omg.org/spec/BPMN/2.0.2/PDF. (Accessed: 15 Feb 2024).

[30] About the business process model and notation specification 2.0. https://www.omg.org/spec/bpmn/2.0/About-BPMN. 2010. (Accessed: 15 Feb 2024).
[31] Camunda. Process orchestration for end-to-end automation. https://camunda.com. (Accessed: 15 Feb 2024).
[32] L. Dikmans, Transforming BPMN into BPEL: why and how. Oracle Middleware/Tech. Details/Tech. Article. *https://www.oracle.com/technical-resources/articles/dikmans-bpm.html*. (Accessed: 16 Oct 2024).
[33] M. Yannakakis, et al., Hierarchical state machines, in: J. van Leeuwen, O. Watanabe, M. Hagiya, et al. (Eds.), Theoretical Computer Science: Exploring New Frontiers of Theoretical Informatics, Springer, Cham, 2000, pp. 315–330. https://doi.org/10.1007/3-540-44929-9_24.
[34] A. Girault, B. Lee, E.A. Lee, Hierarchical finite state machines with multiple concurrency models, IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. 18 (6) (1999) 742–760, https://doi.org/10.1109/43.766725.
[35] C.A.R. Hoare, Communicating sequential processes, Commun, ACM 21 (8) (1978) 666–677, https://doi.org/10.1145/359576.359585.
[36] C. Cassandras, Discrete Event systems: Modeling and Performance Analysis, 1st ed., CRC, Boca Raton, FL, 1993.
[37] O. López-Pintado, M. Dumas, L. García-Bañuelos, et al., Controlled flexibility in blockchain-based collaborative business processes, Inf. Syst. 104 (2022) 101622, https://doi.org/10.1016/j.is.2020.101622.
[38] C. Di Ciccio, A. Cecconi, M. Dumas, et al., Blockchain support for collaborative business processes, Inf. Spektrum 42 (3) (2019) 182–190, https://doi.org/10.1007/s00287-019-01178-x.
[39] A. Bagozi, D. Bianchini, V. De Antonellis, et al., A three-layered approach for designing smart contracts in collaborative processes, in: H. Panetto, C. Debruyne, M. Hepp (Eds.), On the Move to Meaningful Internet Systems: OTM 2019 Conferences, Springer, Cham, 2019, pp. 440–457. https://doi.org/10.1007/978-3-030-33246-4_28.
[40] A. Mavridou, A. Laszka, Designing secure Ethereum smart contracts: a finite state machine based approach, in: S. Meiklejohn, K. Sako (Eds.), Financial Cryptography and Data Security, Springer, Berlin, Heidelberg, 2018, pp. 523–540. https://doi.org/10.1007/978-3-662-58387-6_28.
[41] A. Mavridou, A. Laszka, Tool demonstration: fSolidM for designing secure Ethereum smart contracts, in: L. Bauer, R. Küsters (Eds.), Principles of Security and Trust, Springer, Cham, 2018, pp. 270–277, https://doi.org/10.1007/978-3-319-89722-6_11.
[42] A. Mavridou, A. Laszka, E. Stachtiari, et al., VeriSolid: correct-by-design smart contracts for Ethereum, in: I. Goldberg, T. Moore (Eds.), Financial Cryptography and Data Security, Springer, Cham, 2019, pp. 446–465. https://doi.org/10.1007/978-3-030-32101-7_27.
[43] J. Jin, L. Yan, Y. Zou, et al., Research on smart contract verification and generation method based on BPMN, Mathematics 12 (14) (2024) 2158, https://doi.org/10.3390/math12142158.
[44] M. Rodler, W. Li, G.O. Karame, L. Davi, EVMPatch: timely and automated patching of Ethereum smart contracts, in: Proceedings of the 30th USENIX Security Symposium (USENIX Security 21), USENIX, 2021, pp. 1289–1306. https://www.usenix.org/conference/usenixsecurity21/presentation/rodler.
[45] Z. Li, Y. Zhou, S. Guo, et al., SolSaviour: a defending framework for deployed defective smart contracts, in: Proceedings of the 37th Annual Computer Security Applications Conference, ACM, 2021, pp. 748–760. https://doi.org/10.1145/3485832.3488015.
[46] H. Jin, Z. Wang, M. Wen, et al., Aroc: an automatic repair framework for on-chain smart contracts, IEEE Trans. Softw. Eng. 48 (11) (2022) 4611–4629, https://doi.org/10.1109/TSE.2021.3123170.
[47] F. Corradini, A. Marcelletti, A. Morichetta, et al., Flexible execution of multi-party business processes on blockchain, in: Proceedings of the 5th International Workshop on Emerging Trends in Software Engineering for Blockchain, ACM, 2023, pp. 25–32. https://doi.org/10.1145/3528226.3528369.
[48] F. Corradini, A. Marcelletti, A. Morichetta, et al., Engineering trustable choreography-based systems using blockchain, in: Proceedings of the 35th Annual ACM Symposium on Applied Computing, ACM, 2020, pp. 1470–1479. https://doi.org/10.1145/3341105.3373988.
[49] G. Falazi, M. Hahn, U. Breitenbücher, et al., Modeling and execution of blockchain-aware business processes, SICS Softw. Intensive Cyber Phys. Syst. 34 (2) (2019) 105–116, https://doi.org/10.1007/s00450-019-00399-5.
[50] G. Falazi, M. Hahn, U. Breitenbucher, et al., Process-based composition of permissioned and permissionless blockchain smart contracts, in: Proceedings of the 2019 IEEE 23rd International Enterprise Distributed Object Computing Conference (EDOC), IEEE, 2019, pp. 77–87. https://doi.org/10.1109/edoc.2019.00019.
[51] P. Klinger, L. Nguyen, F. Bodendorf, et al., Upgradeability concept for collaborative blockchain-based business process execution framework, in: Z. Chen, L. Cui, B. Palanisamy, et al. (Eds.), Blockchain – ICBC 2020, Springer, Cham, 2020, pp. 127–141. https://doi.org/10.1007/978-3-030-59638-5_9.
[52] J. Poon, Plasma : scalable autonomous Smart contracts, https://www.semanticscholar.org/paper/Plasma-%3A-Scalable-Autonomous-Smart-Contracts-Poon/cbc775e301d62740bcb3b8ec361721b3edd7c879, 2017. (Accessed: 1 Jan 2023).