Blockchain: Research and Applications 4 (2023) 100115

Contents lists available at ScienceDirect

Blockchain: Research and Applications

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

Research Article

TABS: Transforming automatically BPMN models into blockchain smart contracts

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

[a] Faculty of Computer Science, Dalhousie University, Halifax, NS B3H 4R2, Canada
[b] Sobey School of Business, Saint Mary's University, Halifax, NS B3H 3C3, Canada

ARTICLE INFO

Keywords:
Blockchain
Business processes modeling notation (BPMN)
Discrete event (DE) modeling
Finite state machine (FSM)
Hierarchical state machines (HSM)
Smart contract
Off-chain computation
Privacy
Smart contract interoperability

ABSTRACT

Research on blockchains addresses multiple issues, with one being the automated creation of smart contracts. Developing smart contract methods is more difficult than mainstream software development as the underlying blockchain infrastructure poses additional complexity. We report on a new approach to developing smart contracts with the objective of automating the process to increase developer efficiency and reduce the risk of errors introduced by software developers. To support industry adoption, we use Business Process Model and Notation (BPMN) modeling to describe an application while targeting applications in the trade vertical. We describe a system that transforms a BPMN model into a multi-modal model that combines Discrete Event (DE) modeling for concurrency with Hierarchical State Machines (HSMs) to represent application functionality. Then, further transformations are used to transform the DE-HSM model into methods in smart contracts. The system lets the modeler decide which of the independent patterns should be transformed into methods of a separate smart contract that is deployed on a sidechain for the purpose of (i) reducing processing costs and/or (ii) providing privacy so that other participants in the smart contract do not have visibility into the processing of the pattern. We also briefly describe a proof-of-concept tool we built to demonstrate the feasibility of our approach.

1. Introduction

Blockchains have received fierce attention since the publication of the Bitcoin white paper in 2008 and the subsequent creation of the Bitcoin blockchain in 2009. Although most public attention focuses on cryptocurrency trading, business, research, and software industry communities have espoused the technology due to its desirable properties of trust, immutability, availability, and transparency, amongst others. Thus, blockchain platforms, accompanying languages, and support tools for writing smart contracts are deservingly getting increased attention. However, as with any new technology, blockchains and their smart contracts also pose new challenges in both blockchain infrastructure and blockchain application development that have slowed down the adoption of the technology.

Blockchain application scalability, transaction throughput, and high cost are some of the main issues that are being addressed by research. To avoid the high cost of storage and computation, Eberhardt and Tai [1] discussed techniques that can be used for off-chain storage and computation to reduce their cost. They categorized methods based on what is moved to off-chain into (i) off-chain storage, (ii) off-chain computation, and (iii) a hybrid approach, in which both storage and computation are off-chained. In Refs. [2,3], computation is performed off-chain while proofing techniques are used to record the results of computation on the blockchain.

Another impediment to blockchain adoption is the high cost of the consensus algorithm used to ensure, by the majority of the blockchain network nodes, that a block of data to be appended to the chain is valid in the face of errors or attacks. For example, Bitcoin's consensus algorithm, which uses a proof-of-work approach, results in a maximum of five to seven transactions per second, while Ethereum's network consensus limits transactions to about 15 per second. Yang et al. [3] assessed the capacity of blockchain networks and showed that it is still a critical problem, and new approaches, such as those utilizing shards [3,4], are being explored for solutions.

Examples of other issues being addressed include the oracle problem due to the blockchain's inability to push or pull data from the external

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

https://doi.org/10.1016/j.bcra.2022.100115
Received 7 December 2021; Received in revised form 26 September 2022; Accepted 9 November 2022

environment, privacy of recorded transactions, and compatibility issues across smart contracts of different blockchains. Comprehensive literature review papers on these other issues have been published with examples being [5-13].

In developing smart contracts, blockchains have complex constraints for applications, one being the maintenance of smart contracts that is challenged by their immutable storage. These constraints mean that it is even more important to use software and security engineering best practices in new approaches and methods for introducing efficiencies, as also recommended in Ref. [9]. We first review research on the development of smart contracts that is closest to our work. We then describe our goal and outline further sections.

1.1. Smart contract development

Smart contracts and their development and maintenance received much attention from the research and development communities, as can be seen by the literature surveys in Refs. [5–13]. Research on smart contracts has concentrated on (a) the analysis and verification of contracts that are written already and (b) approaches to create smart contracts from some higher-level model. In Ref. [7], research on smart contracts is classified into categories of (i) smart contract testing, (ii) smart contract code analysis, (iii) smart contract code metrics, (iv) smart contract security, (v) application performance measurements, and (vi) blockchain applications. Since blockchains are immutable and smart contracts are stored on the blockchain, fixing blockchain bugs in smart contracts easily is an open issue. Hence, we observe a lot of research in the categories listed above that concentrates on analyzing existing smart contracts for security, vulnerabilities, correctness, and the presence of errors.

Khan et al. [6] surveyed and classified research on blockchain smart contracts into two broad categories: *improvement* and *usage*. The usage category reviews how blockchains are used by applications of various types, while the improvement category includes research and tools to improve smart contracts either when writing them or once they exist. Approaches listed in the survey on the writing of smart contracts rely on defining a programming language, for instance, domain-specific or contract-oriented, that facilitates the creation of smart contracts with desirable properties. These approaches, although successful in that they lead to smart contract programs with formally proven safety properties, have not been adopted in practice as they exhibit perceived “complexity” by developers who ultimately determine how well-accepted a programming language is.

An extensive literature survey of close to 200 papers on formal specification and verification of smart contracts was provided in Ref. [13], in which formalism taxonomy was used to classify work into contract-level and program-level categories, depending on the level of abstraction used in modeling and analyses. Further classification was performed depending on the type of abstraction used for the analysis. The contract-level category was further classified into methods using (i) process algebras, (ii) state transition systems, and (iii) set-based methods. Furthermore, the program-level category was classified into (i) abstract syntax trees, (ii) byte code, (iii) control-flow graphs, and (iv) program traces.

The literature survey in Ref. [11] reviewed existing Model Driven Engineering (MDE) approaches to the creation of smart contracts. One of the classifications offered was based on modeling techniques that included Unified Modeling Language (UML), Business Process Modeling Notation (BPMN), Entity-Relationship Diagram (ERD), DEMO modeling tool, ArchiMate modeling tool, and Eclipse Modeling Framework (EMF). Our work is closest to research that uses BPMN for business process modeling, which is then transformed into a smart contract. Early work included (i) Bragagnolo et al. [14] described how UML, BPMN, and ER models could be used to model blockchain-based applications but offered no concrete proposals, and (ii) Mendling et al. [15] described challenges arising when analyzing the blockchain technology using Business Process Management (BPM) lifecycle as a framework. They also identified a number of issues beyond the BPM lifecycle management. Their analyses led them to present seven major research directions that will emerge in the future.

Weber et al. [16] proposed an architecture for creating smart contracts from a distributed application specified as a BPMN model. The transformation process has two phases: design and execution. In the design phase, the BPMN model is analyzed and transformed into a distributed application that uses a blockchain smart contract. There are two components to the system, an off-chain component that accepts input from the external application and transforms it into the calls to the methods of a smart contract, and the on-chain component that includes the smart contract methods and a run-time monitor/mediator that manages the choreography of task activities represented by BPMN elements. From this work, there were two significant follow-up projects, Caterpillar [17] and Lorikeet [18].

Caterpillar has been further augmented with the ability to bind actors to roles dynamically using a proposed policy specification language based on Petri net semantics [19,20]. The proposal has been incorporated into the Caterpillar project software, and the approach was evaluated with positive results. The policy specification is transformed into methods of smart contracts used for enforcement. Di Ciccio et al. [21] also supported model-driven engineering using BPMN for the generation of the methods of smart contracts, and for demonstration of feasibility, they used the Caterpillar [17] and Lorikeet [18] projects.

Loukil et al. [22] also started with a BPMN model, but one of their requirements is to address the issue of smart contract immutability and hence require the creation and deployment of a new contract any time new updates are required. Thus, instead of generating a compiled version of a smart contract from a BPMN model, they advocated the creation and deployment of one generic smart contract that invokes predefined functions. They proposed a system architecture that has three layers: Conceptual, Data, and Flow layers. The conceptual layer performs a set of activities that invoke functions of the data layer.

Lu et al. [23] used business process modeling to support fungible and non-fungible asset management. In addition to BPMN modeling, their architecture also includes asset registry modeling for each fungible and non-fungible asset. For evaluation, they used Lorikeet for BPMN modeling of selected use cases from government and industry, wherein the use cases include fungible/non-fungible asset registration, escrow for conditional payment, and asset swap.

In a research report [24], the authors took a different approach to transforming a BPMN model into a smart contract. Instead of starting with a BPMN collaboration represented by a BPMN diagram, as does most of the other research on transforming BPMN models into smart contracts, they started with a BPMN choreography, which describes conversations amongst the participants but not the details of messages exchanged during the conversation. BPMN conversations within a choreography are not deterministic, as there may be many collaborations within a conversation that satisfy the conversation specification. Thus, the research work focuses on optimizing each conversation and its synthesis.

We also mention work in Refs. [25,26], in which a smart contract is first modeled as a Finite State Machine (FSM) created using a graphical editor. The FSM is transformed into a set of methods of a smart contract, and then each method is examined using their FSolidM tool to amend it with security code to plug the known security holes. Furthermore, the security code cannot be amended by the developer. The issue is that the smart contract needs to be represented as a single FSM first, which limits the usefulness of the approach. However, once the smart contract methods are created, the FSolidM tool can be used to plug the security holes.

1.2. Goal and approach

The main goal of our research is to transform a business application, expressed by a BPMN model, into a set of methods of smart contract(s). The main distinction of our approach to other work that transforms BPMN to a smart contract is that we use Discrete Event–Hierarchical State Machine (DE-HSM) modeling, in which concurrency is modeled using Discrete Event (DE) modeling and functionality is expressed using Hierarchical State Machine (HSM) modeling. As HSM is a form of an FSM, one of the basic modeling techniques in computer science, there is rich theory and many tools that have been developed to reason about FSMs.

However, most importantly, using DE-HSM modeling enabled us to create methods that identify patterns, DE-HSM sub-models, that are suitable to be processed on sidechains. We exploit this property to facilitate:

- Cost reduction by allowing the modeler to select patterns, identified as DE-HSM sub-models, to be processed on sidechains that are cheaper than the mainchain.
- To achieve privacy by processing specific collaboration activities on a sidechain where data are visible only to the collaboration participants as opposed to all actors accessing the smart contract.

1.3. Related work and contributions

Closest to our work is the research on transforming a BPMN model into methods of a smart contract with minimal intervention from the developer [16–23] that was already discussed in a previous subsection. They transformed a BPMN model into a set of tasks that are executed under the control of a monitor/mediator, which is a part of the smart contract deployed on a blockchain. The off-chain part of the system transforms the application's input into calls to a smart contract. Input is parsed to determine application's input, and choreography (not BPMN terminology) is used to determine which task is to be executed. Incorrect sequences of input form “non-conformant” input streams that need to be identified by the system, while correct sequences should result in the execution of correct tasks.

We depart from the related work in several major aspects that also form our **contributions**:

1. We use DE-HSM multi-modal modeling that transforms a BPMN model into a DE-HSM model in which sub-models are FSMs and DE modeling is used to model time.
2. We support the automatic transformation of selected DE-FSM sub-models into the methods of a smart contract. The choreography of collaboration tasks is represented by a DE-HSM model consisting of DE-FSM sub-models.
3. Furthermore, the DE-HSM model representation of the BPMN model into individual DE-FSM sub-models is such that each individual model is an independent component, also referred to as a pattern, with a desirable property that once the pattern execution starts, it is localized until execution leaves that pattern. This property is exploited to facilitate off-chain processing of such patterns and to support privacy.
4. The modeler provides information on the patterns/sub-models and selects those to be processed off-chain. The Transforming Automatically to Blockchain Smart Contracts (TABS) system transforms the DE-HSM model into smart contracts, one deployed and executed on the mainchain and the other deployed and executed on a sidechain (we currently support off-chain processing on sidechains only). The mainchain contract invokes the methods of the sidechain contract that implements the patterns that were selected by the developer to be executed on a sidechain. The mainchain may be either one of Ethereum (private or public) and Hyperledger Fabric, while the sidechain may be one of Quorum (Ethereum-EVM sidechain) or the Hyperledger Fabric channel.
5. The use of DE-HSM modeling led to automated support for sidechain processing that is used (i) to reduce the cost of processing by moving computation from the mainchain to a cheaper sidechain and (ii) to achieve privacy in collaboration, wherein data required for use in a pattern that is processed on a sidechain is private and hence is inaccessible to actors who do not participate in collaborating on that pattern.

1.4. Outline

In Section 2, we provide background. Section 3 overviews our architecture and also presents assumptions and the BPMN modeling features that our system currently supports. Section 4 describes (i) the transformation of a BPMN model into a DE-HSM model and then (ii) the creation of DE-FSM sub-models. Section 5 describes how the DE-HSM model is transformed into the methods of a smart contract(s) and how the modeler is provided with the choice of deploying sub-models for processing off-chain to (i) decrease the cost of processing by performing work off-chain where processing is less costly and/or (ii) provide privacy. Section 6 describes our proof-of-concept (PoC) system, the TABS tool, and its evaluation. Section 7 offers concluding remarks.

2. Background

In this section, we first briefly review the trade finance use case and its properties. We then describe DE-FSM multi-modal modeling that combines DE modeling for concurrency with FSM modeling. We conclude with an overview of BPMN elements.

2.1. Trade finance applications

Blockchains and their smart contracts are not suitable for all types of applications. Smart contracts are not suitable for applications that exhibit heavy computational requirements, as the execution of smart contract transactions requires validation involving all or a number of the blockchain nodes. As their name implies, smart contracts are particularly suitable for applications that involve the collaboration of different actors, possibly of different organizations, with varying requirements on information availability, security, and privacy, amongst others. And trade applications involve activities that exhibit those properties and thus should be suitable for blockchain applications, as is confirmed by the much higher adoption of blockchains in trade, which also includes finance in trade and hence also Decentralized Finance (DeFi). Thus, trade and finance applications are our target area, which is a broad area involving third parties for the purposes of efficient processing of transactions by participants, such as banks, customs, shippers, and insurance companies.

2.2. FSMs, Hierarchical State Machines, and Nested Multi-modal Modeling

FSM modeling has been used frequently in the design and implementation of software. Adaptations of FSM modeling techniques were proposed to expand the applicability of FSMs to modeling applications. For instance, a guard along an FSM transition is used to specify a Boolean condition on the state's variables that must be true for the transition to take place. In the late 1980s, FSMs were extended to address the issue of reuse of patterns with the concept of hierarchy, leading to HSMs that may contain states that are themselves other FSMs. Harel [27] showed that FSMs can be combined hierarchically: A single hierarchical state at one level can be considered to be in several states concurrently, as represented by an FSM(s) in a lower level of the hierarchy, and FSMs may also be combined, leading to concurrent FSMs.

An HSM can be defined using induction as follows (due to Ref. [28]

and also described in Refs. [27,29], and others): In the base case, an FSM is a hierarchical machine. Suppose that M is a set of HSMs. If F is an FSM with a set of states, S, and there is a mapping function f: S-> M, then the triple (F, M, f) is an HSM. Each state, $s \in S$, that represents an HSM is replaced by its mapping $f(s)$. HSMs recognize the same language as their corresponding flattened FSM. However, HSMs do not increase the expressiveness of FSMs, only succinctness in representing them.

Girault et al. [28] described how HSM modeling can be combined with the concurrency semantics of a number of concurrency models, including communicating sequential processes [30] and DEs [31]. They described how an HSM model can represent a module of a system under a concurrency model that is applicable only if the system is in that state. This enables the representation of a subsystem using a particular concurrency model that may be nested within a hierarchical state of a higher-level FSM. This may be used in multi-modal modeling, in which different (hierarchical) states may be combined with different concurrency models that are best suitable for modeling concurrent activities for that particular state. We exploit the concept of multi-modal modeling to allow the designer to model concurrent but independent activities by concurrent FSMs at the lower level of hierarchy in order to avoid the state explosion that may arise in such situations [28]. In Refs. [32,33], we showed that a multi-modal model that combines DE modeling with FSM modeling may be used to model trade finance applications; such a combination of models is referred to as DE-HSM multi-modal modeling. We also showed that it is possible to automatically transform a DE-HSM model of such an application into methods of a smart contract deployed on a blockchain.

A DE-HSM has external inputs, and it produces outputs. The model represents how external inputs form inputs to sub-models and how those sub-models are interconnected to produce the final output. However, if the sub-model's interconnection is such that there are no loops, then the model can be viewed as a zero-delay model [29], in which the individual DE queues may be combined into one DE queue.

Feedback loops in a DE-HSM arise due to feedback loops in the BPMN model from which the DE-HSM model is derived. None of the approaches to producing smart contracts from a BPMN model that were discussed in the introductory section addressed the issues of the feedback loops in a BPMN model. Feedback loops cause various difficulties, such as deterministic semantics, and will be discussed further in a later section.

In modeling, the term DE-FSM modeling refers to a model for which it is known that it does not contain hierarchical states.

2.3. BPMN overview

BPMN is viewed as a de facto standard for describing business processes. It was developed by the OMG organization with the objective of BPMN to be understandable by all business users, from business analysts, through technical developers implementing the processes, to people managing those processes. That it has been adopted in practice is demonstrated by many available software platforms that provide for the modeling of business applications with the objective of automatically creating an executable application from the BPMN model.

For instance, Oracle Corporation uses the Oracle Business Process Analysis Suite to model business processes using BPMN and transform a BPMN diagram into a blueprint for executable Business Process Execution Language (BPEL) processes [34], wherein the blueprint represents the logic of the application in terms of concurrent processes and their interactions, while details of individual tasks are supplied by implementors. Another example is the Camunda software platform, which is also used to develop a BPMN model that is then transformed automatically into a Java application. Our approach is similar, but instead of targeting BPEL or Java as the 'target' for creating executable applications, we target smart contracts executing on blockchains.

The BPMN standard categorizes its elements into five basic categories: (i) Flow objects, (ii) Data, (iii) Connecting objects, (iv) Swimlanes, and (v) Artifacts.

2.3.1. Flow objects

They include Events, Activities, and Gateways.

- **Events:** There are three types of events: *Start*, *Intermediate*, and *End*. Start and End events are special in that they denote the start and end of an activity or a business process. Intermediate events include many different types with examples that include message timer, timer, conditional, escalation, and multiple parallels. Each type has a special symbol with an icon having special markings to indicate properties, such as throwing or catching, interrupting and non-interrupting, and boundary. Some of the throwing events are connected to catching events with sequence flow to show the flow of control.
- **Activities:** As the name implies, activities represent work performed in a business process flow, activities that include *Tasks*, *Sub-processes,* and *Call Activities*. Call activities allow the re-use of Tasks and Sub-processes and thus enable their nesting. An activity may be performed once or many times, in which case it is a looped activity. A looped activity is a specified activity that is intended to be performed on each element of a set of objects that may be viewed as input to the looped activity. The looped activity may be either Standard, wherein the specified activity is executed sequentially once for each object in the input set of objects. Or it may be a Multi-instance activity, in which case the activity is executed concurrently on each object in the set of objects.
  - Task is an activity that is independent of other activities or BPMN elements.
  - Sub-process is a compound activity that can be represented by other elements of which the scope is limited to the sub-process itself. A sub-process may be defined further as a transaction, in which case the sub-process must be supported by a special protocol that ensures that all of the transaction activities have been completed successfully and that all activities completed partially by a sub-process are "undone" by performing compensating activities. Compensating activities are shown using a Compensating association.
- **Gateways:** They are used to represent the flow of a business process. A gateway may (i) have one or more incoming/converging flows to form a join/merge gateway and (ii) have one or more outgoing/diverging sequence flows, in which case it is a split/fork gateway. If a gateway has multiple incoming and multiple outgoing sequence flows, then it is both a converging and a diverging gateway.
  - A diverging exclusive gateway, which may have conditions specified on the outgoing paths and permit a default outgoing path, allows only one outgoing execution at a time.
  - An inclusive diverging gateway allows many outgoing sequence flows to be active. If an outgoing flow has an associated condition, then that condition must evaluate to be true at run-time in order for a flow to proceed along that path.
  - A parallel converging gateway does not have conditions on its path: Each incoming sequence must receive a token (an incoming flow must arrive on a path) before triggering the outflow. Once an outflow is triggered, all outgoing paths are activated and proceed in parallel if the gateway is also diverging.
  - An inclusive converging gateway passes the token only if a token has arrived on each of the enabled incoming paths; otherwise, there is a wait until the tokens do arrive.

2.3.2. Data

Included are *Data Objects* that provide information about what data activities use or produce. A Data Object is used to represent either an object or a set of objects. *Data Input* provides input information to processes, while *Data Output* is used to receive information from processes. A *Data Store* use is self-explanatory.

2.3.3. Connecting objects

Connecting objects are used to indicate the sequence of execution.

Connecting objects can be (i) Sequence Flows, (ii) Message flow, (iii) Associations, and (iv) Data Associations.

- Sequence flows show the normal flow. In addition, there are specialized types of sequence flows, namely Conditional flow that has a condition that is evaluated at run-time to determine if the flow may proceed and Default and Exception flows that are self-explanatory.
- Message flow shows the flow of messages.
- Association, such as Text association, is used to link information and Artifacts with BPMN graphical elements.

2.3.4. Swimlanes

A swimlane represents a participant in a business process. There are two types of swimlanes: *Lanes* and *Pools*. A pool with lanes represents an organization; while lanes within a pool represent individual participants within that organization. Only message flows can cross pools; no other BPMN elements can do so.

2.3.5. Artifacts

They are used as a form of documentation to provide additional information about a process. The two standardized Artifacts are *Group* and *Text Annotation,* but modelers are free to add as many Artifacts as necessary. Group provides a graphical representation of a collection of elements. Text Annotation is used to provide additional textual information to a BPMN diagram element.

In version 2.0.2, OMG introduced choreography in BPMN to represent the way business participants coordinate their interactions through conversations. Choreography is a concept at a higher level of abstraction than the core BPMN elements. It hides many details that are required for the elaboration of a BPMN model into an executable application, regardless of whether it is blockchain-based or not. In a BPMN model, choreography focuses on showing message exchanges amongst the participants, however, without providing the details on the messages exchanged or their content. In a BPMN choreography, instead of showing detailed message exchange, commands of higher-level abstraction are used to represent the exchange of messages using a single request-reply type element, and the BPMN choreography is thus unsuitable for modeling an application to be transformed into a smart contract.

3. System architecture

Although we discuss our assumptions throughout the paper, we enumerate them collectively in this section. They include assumptions on the BPMN model and which BPMN elements we support. We then provide a high-level overview of the architecture that consists of two components: One used in the design phase to transform a BPMN model into the methods of smart contract(s) and one used during the run-time phase to monitor and manage the smart contract deployed on a blockchain. Each phase is briefly overviewed and then described in more detail in subsequent sections.

3.1. Assumptions

We first describe assumptions we make regarding a BPMN model and then assumptions made on the system architecture and its implementation.

3.1.1. Assumptions on the BPMN models

These assumptions deal primarily with loops, sub-processes, converging gateways, model preprocessing, and which BPMN elements we currently support in the TABS tool.

a) *Assumptions on parallel and looping sub-processes*

That loops in BPMN models cause difficulties with BPMN semantics has been recognized and examined by a number of researchers. For instance, Dijkman et al. [35] analyzed the semantics of the BPMN model and identified a number of issues arising due to the inexact nature of the semantics in the BPMN specifications. And one of the issues identified was difficulties arising due to the uncertain semantics of the BPMN specification for feedback loops. We do support a limited form of looping in BPMN diagrams, wherein the limitation is that a loop is defined only for patterns, that is, portions of a BPMN diagram, that can be represented by sub-processes. That is, the BPMN elements of a pattern are connected to the rest of the diagram only by two connecting objects, such that one is incoming and one is outgoing. In effect, such a portion/pattern of the BPMN diagram can be represented by a sub-process, as it also has only one incoming and one outgoing connecting object. In fact, the way we handle such a pattern with a feedback loop is by transforming it into a looping sub-process, but with limitations on the looping conditions, as shall be described under the looping sub-process description below. Interestingly, none of the other approaches to producing smart contracts from a BPMN model, which we discussed in the introductory section, made any mention of how loops in a BPMN model were handled.

We support sub-processes in a straightforward way by replacing each sub-process with its BPMN model of core elements (BPMN diagram of the expanded sub-process) before transforming a BPMN model into its DE-HSM representation. We also support limited forms of *parallel* and *looping sub-processes*.

Parallel sub-process: Our approach is to replace a BPMN sub-process with its elaboration in terms of its activities expressed using the individual BPMN elements when the sub-process is expanded. To support parallel streams for parallel execution, our approach needs to know the maximum number of parallel activities in a process, as we need to create the required number of streams, one for each parallel activity, before the smart contract deployment. Before transforming the DE-HSM model into a smart contract, the modeler needs to specify the maximum number of parallel streams for a parallel sub-process. That information is then used to create the maximum number of parallel streams, which exist in the smart contract whether they are used or not. Thus, we do not fully support the BPMN semantics in that there may be an arbitrary number of separate parallel streams that are used to process a collection input into a sub-process, wherein the collection has an arbitrary number of elements that results in an arbitrary number of parallel streams, each one to process one element of the collection. However, this issue arises generally with handling looping mechanisms in deterministic systems, such as systems expressed in FSM or HSM modeling.

Looping sub-process: We do support a limited form of a looping sub-process, wherein the limitation is that the maximum number of iterations is known before the start of the execution of a looping sub-process. We model a sub-process without its looping property, and we accommodate the looping after modeling, before the creation of a smart contract in a way that is similar to the usual *for-loop* programming construct. Transformation prepares inserts the script for a looping variable just before entering the looping sub-process, with the looping variable initialized based on the size of the input collection. Furthermore, the translator inserts the script at the end of the looping sub-process to increment the looping variable and check whether to perform the loop again.

b) *Assumptions on converging gateways*

Converging gateways may be parallel, in which case for a token to be passed, a token must have arrived on each of the incoming edges of a converging gateway.

However, the converging gateway may also be inclusive, in which case the BPMN semantics state that when a token arrives on an incoming edge to an inclusive converging gateway, the token can be passed only if a token has arrived on each of the incoming pathway that is enabled; otherwise, there is a wait for the remaining tokens arriving on such enabled pathways. However, we make a simplification for an inclusive converging gateway in that we simply pass the token; we check the other pathways neither for enablement nor for a token arrival.

c) *Assumptions on BPMN model preprocessing*

Once a BPMN model is created and expressed using its standard XML representation [35], we assume that the model is transformed into an equivalent BPMN model that is well-formed, wherein the well-formed BPMN model is defined as in Ref. [35]: A BPMN model is well-formed in it satisfies the following conditions: (i) there is only one start event; (ii) there is only one end event; (iii) fork/split decision gateways have one incoming flow and more than one outgoing flow; (iv) join/merge gateways have one outgoing flow and more than one incoming flow; and (v) there are no data-based splits/forks or joins/merges—they are expressed using equivalent gateway constructs. The above conditions are not restrictive as any BPMN model can be transformed into an equivalent well-formed BPMN model as described in Ref. [35].

The proposed TABS tool, described in this paper, allows users to compose a BPMN diagram directly on the canvas by invoking an API provided by Camunda [36]. The model is saved by the tool in the BPMN XML representation [37] and then transformed into a DE-HSM model by our TABS tool. The BPMN model may contain participants (Pool and Lane), activities (tasks, sub-processes, and transaction), gateways, and events (default, start, end, message, and timer) [38]. Fig. 1 shows the symbols that our TABS tool supports, albeit as noted in Section 3.1.1(a), we impose limits on looping and parallel sub-processes and arbitrary loops in BPMN diagrams.

3.1.2. Other assumptions

a) *No support for long-term trade transactions*

BPMN has the concept of a transaction defined on a sub-process, and a blockchain also supports transactions. That is, a sub-process can be defined in BPMN as an atomic transaction in that the whole sub-process needs to be either completed entirely or none of its effects, such as updates to the state variables, be recorded on the blockchain. Any activity that was performed by a sub-process that was subsequently aborted must be "undone" by compensating activities. Blockchains also have transactions of which semantics are defined in terms of blocks of transactions. Furthermore, trade applications also have the concept of a transaction, wherein transactions may be defined for a group of activities that may span long periods of time and may apply to data that is a subset of the data processed by a trade application. The problem is that these three concepts of a transaction, one for the trade vertical, one for BPMN, and one for blockchains, are mismatched as they do not align, which is akin to the impedance mismatch between the relational and object-oriented DBs

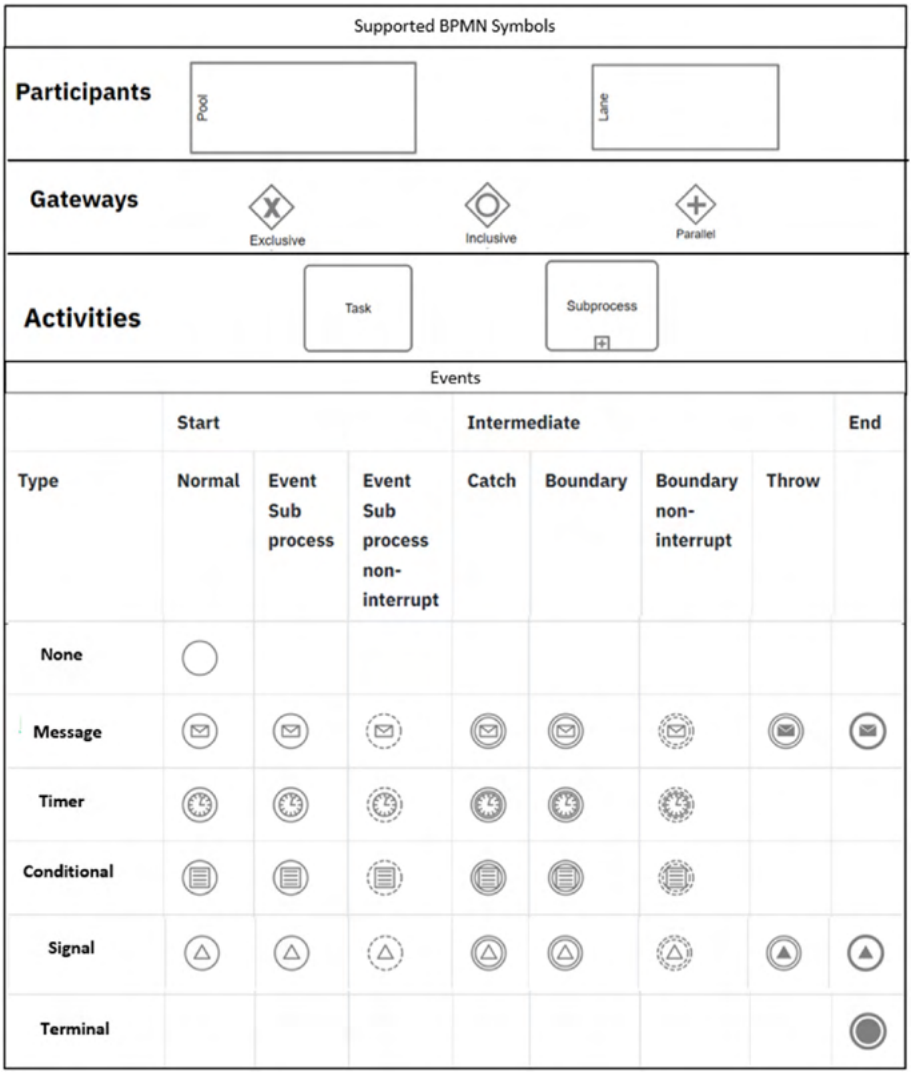


Fig. 1. BPMN (business process model and notation) symbols supported by TABS (transforming automatically to blockchain smart contracts).

[39]. We do not address this mismatch amongst the BPMN, blockchain, and trade transactions in this work, but we do plan on doing so in our near-future work.

b) *Off-chain storage*

It is well known that storing objects or data on a blockchain is expensive, particularly for a public blockchain, such as Ethereum. It is thus a standard practice to store large objects/data off-chain and store on the mainchain only the hash code of the data stored off-chain. Furthermore, the hash code of an object is used any time the object is retrieved by a smart contract to ensure the immutability property of an object stored off-chain. Thus, we also assume that any objects are stored off-chain. An exception to that is any object used in support of the monitor of the architecture shown in Fig. 2. We facilitate off-chain storage by using IPFS. IPFS is a distributed system for storing and accessing files, websites, applications, and data [40]. It is resilient and has decentralized control. Due to its resiliency, decentralized control, and content addressing using a hash code, it is particularly suitable as off-chain storage for blockchains. IPFS storage is, content addressable using the hash code of an object.

c) *BPMN task element*

The BPMN task element represents a self-contained task. However, our system does not provide the script for the task element, the developer does. Our system does prepare the method skeleton, including the method's signature and the method's invocation, while the developer is asked to supply the code for each task before deployment. This approach is standard when creating applications from a BPMN model. For instance, Oracle Corporation uses the same approach in its Oracle Business Process Analysis Suite when transforming a BPMN diagram into a blueprint for executable BPEL processes [41], and a similar statement also applies to the Camunda platform, which also asks the modeler to provide the script for each of the PBMN model's tasks.

3.2. Architecture Overview

The architecture of the system is shown in Fig. 2, which shows two major phases: Design and Deployment/Installation phase, and the

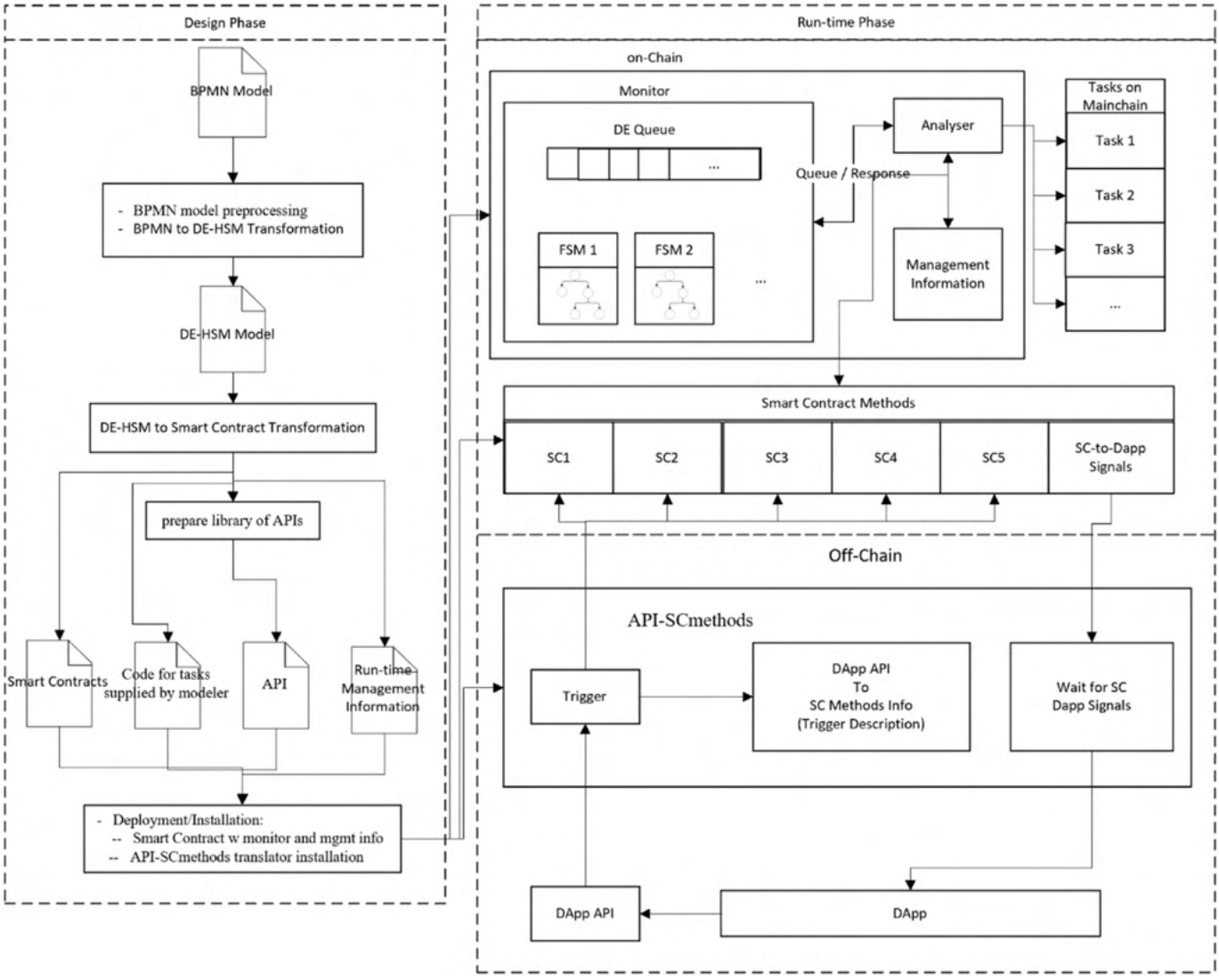


Fig. 2. Architecture overview.

Execution/run-time phase. In the first phase, the BPMN diagram is first transformed into the DE-HSM model and then into the methods of a smart contract. However, besides the creation of the smart contract methods with supporting run-time management information, the transformation also prepares API methods to be used by a Distributed Application (DApp) to interact with the methods. Once the smart contract is prepared, it is deployed in the Design and Deployment/Installation phase, which deploys the smart contract on the blockchain and also installs the API and associated methods in a network node so that a DApp is able to communicate with the smart contract. The smart contract includes a monitoring module with run-time management information stored in the data structures prepared during the design phase. They include the DE queue of events and FSM definitions corresponding to the DE-FSM sub-models and FSMs to support concurrent processing within individual DE-FSM sub-models.

After the smart contract deployment and installation of API methods, run-time processing involves the user interacting with the DApp that invokes appropriate API methods in response to user input. This input is examined by the API-SCmethods module that translates it into an invocation of an appropriate smart contract method.

A smart contract method input is passed to the Monitor that:

- Accepts inputs from a smart contract method, analyzes them, and queues an appropriate event into the DE queue as a response to the input.
- Removes events from the DE queue one at a time, and for each event:
  - o Analyzes an event's information and invokes an appropriate FSM while passing it input derived from the event's information.
  - o The invoked FSM takes the input, examines its state, and fires causing a transition while generating output that is returned back to the monitor.
  - o The monitor analyzes the output produced by the FSM's firing and may perform one or more of the following actions:
    - Invoke one of the tasks
    - Queue an event into the DE queue
    - Respond to the smart contract invocation
    - Cause a signal to be delivered to the DApp

3.3. Design and Deployment/Installation phase

The design phase consists of the following main steps:

1. Initialization and transformation of the BPMN model into the DE-HSM multi-modal model.
2. Transformation of the DE-FSM sub-model into a set of methods of a smart contract.
3. Deployment and installation: Deploy the smart contract(s) and install the API-SCmethods and the DApp API.

As was already indicated in the previous section, our architecture is similar to that of Refs. [16–18], and for the sake of comparison, we use the same terminology as in Ref. [16], which has also been used in other work. Fig. 3 shows a BPMN model to be transformed by a Translator of the Design phase into a DE-HSM model, which is followed by producing the smart contract methods. Similarly, the run-time phase involves a mediator/monitor that takes external events/triggers, which the DApp delivers to the run-time methods as parameters when calling them. The response is delivered to the DApp via output parameters upon return.

In the initialization of the design phase, data structures are prepared for the subsequent design steps that will include the information that is used during the execution phase. After initialization, transformation from the BPMN model into the DE-HSM model deals with determining

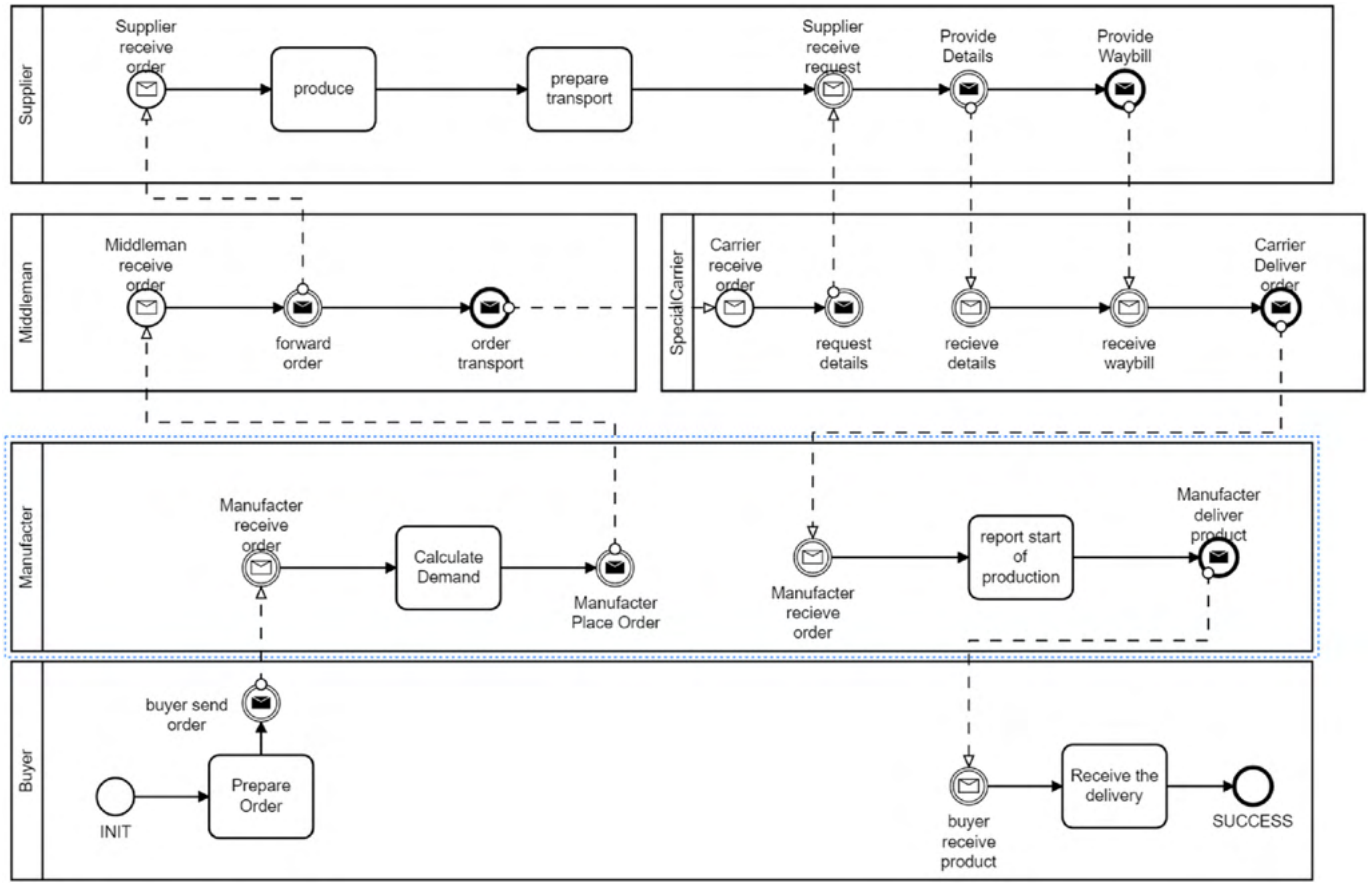


Fig. 3. A BPMN (business process model and notation) model for supply chain example.

which parts of the BPMN model should be represented by which of the DE-FSM sub-models. Each of the DE-FSM sub-models includes a DE model to represent concurrency and an FSM model to represent functionality. Choreography (not a BPMN term), which orchestrates the execution of activities in possibly concurrent processes, is represented by a DE-HSM model that represents the control flow amongst the DE-FHM sub-models, as will be described later. Furthermore, each DE-FSM model may also have internal parallel activities if the BPMN collaboration it represents has parallel or inclusive gateways.

In addition to the smart contract methods and information for the run-time Monitor, the design phase also prepares API methods to be invoked by the DApp. The API methods presented to the DApp are focused on activities amongst actors and thus roughly correspond to the flows of objects between either pools or swimlanes. As a consequence, when input is received via a message from one actor to another, receiving a message will result in a call to the API with input being the content of the received message. The Design phase prepares appropriate information for the API-to-SCmethods translator that assists in determining which of the smart contract methods should be invoked and which input parameters to the API should be included as input to the smart contract method invocation.

The last steps are relatively straightforwards, as they include deployment of the smart contract methods and installation of the DApp API and API-to-SCmethods modules on network nodes.

3.4. Execution phase

The execution phase includes two phases: Initialization and Execution.

Initialization deals with issues, such as the authentication of the actors, also referred to as participants. Once the smart contract is deployed, execution is managed by the Monitor module.

It should be noted that the activities described above need to be performed by any approach that generates smart contract methods from a BPMN model. Consequently, parts of the architecture for the run-time phase are similar to other architectures, such as the architecture in Refs. [16–18]. For that reason, we use the same or similar terminology, including terms of monitor/mediator and triggers. Fig. 2 shows, for the run-time phase, an off-chain DApp API module that provides an API to the DApp. The module also shows an API-to-SCmethods component that transforms an API call into a call to a smart contract method. In other research, this component is called “triggers”, but we feel that the API-to-SCmethods name is more appropriate than triggers.

The DApp API methods correspond to information flow across pools or swimlanes, and when a connecting object crosses a pool or a lane, information about the origin and destination of the connecting object is included in input parameters to the API-to-SCmethods component to determine which smart contract method is to be invoked.

A smart contract, at times, needs to inform external actors of certain events to which a user should react. For instance, a contract may go through computation along different paths that eventually meet, at which point a user(s) may need to be informed of the results before the computation proceeds. Our systems facilitate this by the API, which is off-chain, having a process that invokes a special API method (in Fig. 2, the process is shown as the Wait for SC to DApp Signals process) that in turn invokes a smart contract method that waits for a “wakeup” event that may be generated by another smart contract method. When a smart contract needs to signal the external DApp, it raises a “wakeup” event that releases the method waiting for a wakeup event, and it can thus return to the DApp information about an event that has occurred within the execution of the smart contract.

When a DApp makes a call to the DApp API, the call information with its parameters is provided to the API-to-SCmethods module that determines which smart contract method is to be invoked, and then it invokes the smart contract method and waits for its response.

The smart contract method provides its information to the Monitor that uses its Analyzer sub-module to examine the input parameters and use them to queue an appropriate event into the DE queue. When that event reaches the head of the queue, it is dequeued and passed to the appropriate FSM for firing. The FSM receives its input, checks the current state, and fires; that is, it makes a transition while generating output. The transition's output indicates if a task or a sub-process, associated with the new FSM state, needs to be performed. For instance, if the transition was to a state that represents a task calculate demand, then a task created for that purpose is invoked. As a part of the FSM transition, the smart contract analyzes the returned output from a transition, and consults information on tasks and sub-processes, and determines which of the following should be performed: (i) schedule another event by queueing it into the DE queue; (ii) prepare output parameters and return them to the smart contract method that invoked them; (iii) produce output and cause a DAapp signal to be generated; and (iv) invoke another smart contract method.

3.5. Example

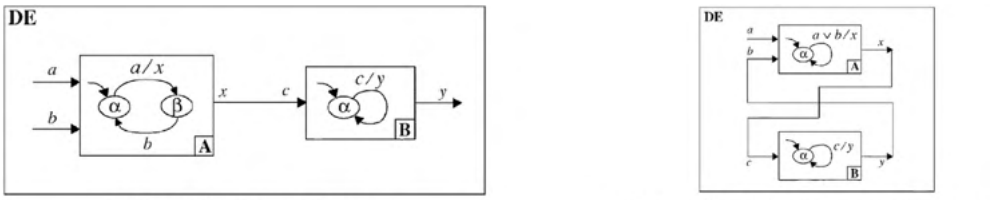
Consider the BPMN diagram in Fig. 3 and assume that processing has progressed successfully and that (i) the manufacturer has sent a message to the middleman with an order for parts; (ii) the message with the order has just been received by the DApp, and that the DApp just invoked the API with information about the received message. The API method takes the information from the API method and packages it as input to the API-to-SCmethods component together with information on the source and destination of the message flow object. The component consults its information, invokes an appropriate smart contract method while passing its input parameters, and then waits for the response from the smart contract method.

The smart contract method passes its information to the Analyzer component of the Monitor. The Analyzer checks its management information and transforms the smart contract input information into an event, which includes parameters that are stored in the DE queue. The component then removes an element from the head of the DE queue, processes it, and repeats until the queue is empty. In this example case, there is only one element that is dequeued from the DE queue, an element representing the reception of the message containing the order.

When the event, which represents the reception of an order, is dequeued from the DE queue, it is processed by the Analyzer by (i) determining which of the FSMs is to be fired to process the dequeued event; (ii) preparing the input parameters for the FSM; and then firing the FSM, i.e., causing the FSM's transition. The FSM takes the input, the current FMS state, and determines the transition while producing output that contains the message content and also information about the new state that corresponds to a task ProcessOrder. Since that state has been reached and it represents the ProcessOrder task, the corresponding task is invoked that produces output from the transition that includes the processed order. As the output from the task, the processed order, is passed to the next element, it is returned to the Analyzer as the transition's output. The Analyzer then examines the output from the transition and checks the new state, which sends a message to the manufacturer, and it determines that the new state is still a part of the same FSM. Consequently, Analyzer creates a new event with input containing the processed order and queues it into the event queue, and the process repeats.

4. Transforming a BPMN model into a DE-HSM multi-modal model and its DE-FSM sub-models

To better understand the transformation process, we briefly review the key properties of the DE-HSM modeling. As our system transforms a BPMN model into a DE-HSM model, we need to identify the DE-HSM sub-models and their interconnections. However, before we proceed, we need to be aware of the model semantics and their implications for modeling. Furthermore, we also explain some of the more subtle points on how certain BPMN elements are represented in the transformation process.



(a) Two child modules refining a parent

(b) Two child modules with loops refining a parent

Fig. 4. Two examples of DE-HSM models (Adopted from Ref. [28]).

Only then do we describe the transformation process itself.

4.1. DE-HSM modeling semantics and loops

We use DE modeling for concurrency combined with HSM modeling for functionality, wherein the modeling is applied hierarchically. As an example, Fig. 4 shows a system that contains two sub-models.

Fig. 4(a) shows a DE-FSM master model containing two DE-FSM sub-models, A and B, and their interconnection. Each of the master and two sub-models has its own queue of events for DE modeling and an FSM for functionality modeling. It should be noted that we referred to the model as a DE-FSM model as the subsystems have FSMs and not HSMs.

In semantics for DE-HSM models, adopted from Ref. [28], an FSM system appears to the DE system as a zero-delay participant/actor. Under the assumption of zero-delay semantics, a system with sub-models is such that the FSM of a child subsystem is the first to fire, that is, it performs the transition on inputs and produces output that is timestamped with the same timestamp as input (hence zero-delay), and only then does the parent system react and make a transition that causes its output. Furthermore, without loops, we can use one global queue of events instead of each subsystem having a separate queue of events [28].

However, loops cause difficulties under the assumed zero-delay semantics. Consider Fig. 4(b) and the two subsystems A and B. There is a loop because output from A forms input to B, and there is another loop in which B's output is input to A. Such loops cause difficulties, and we cannot apply zero-delay semantics in such a case, as execution may never terminate. Consequently, representing a BPMN model using DE-HSM modeling, we wish to avoid such feedback loops due to the semantic difficulties described above. In Section 3.1, we already assumed that a BPMN model does not include any loops that cannot be represented by a looping sub-process. Furthermore, the mechanism for a looping sub-process is implemented as a part of the transformation from the DE-FSM model into the methods of a smart contract that occurs after the DE-HSM model is created from a BPMN model.

4.2. Initialization (in design phase)

In the initialization phase, the translator first transforms a given BPMN model into an equivalent model that is well-formed, as described in Section 3.1.

In general, a BPMN specification does not require the modeler to provide a description of all the information, which is associated with the BPMN model that is required for execution. However, since our objective is to produce executable smart contracts, the TABS system must obtain such information from the modeler. Thus, in the initialization phase, the software interacts with the modeler to ensure that the transformation has information about all data flowing along the edges of a BPMN graph to ensure that the transformation has all the necessary information about guards on gateways and the content of messages. During initialization, we ask the modeler to annotate the BPMN graph accordingly using data associations. The issue is that the BPMN specification does not insist on the modeler providing such information, and hence, all approaches that strive to create an executable smart contract from a BPMN model need to seek information from the modeler if it is missing in a BPMN model [36].

Once the modeler supplies information on the data flowing along the connecting objects, the system asks the modeler to provide the code to be executed by each individual task. Each task has a single input and a single output, wherein each may be a complex object. The tasks are incorporated by the system into the smart contract methods so that they can be executed according to the control flow as specified by the BPMN model.

4.3. Representing BPMN elements for DE-HSM modeling

The transformation of a BPMN model into a DE-HSM model is based on a graph representation of a BPMN model given its BPMN diagram description using XML. For the DE-HSM modeling and the transformation process, we use a Directed Acyclic Graph (DAG) representation of the BPMN diagram. In the DAG representation, each BPMN flow element/object is represented using a graph vertex, while connecting objects, that is, sequence and message flows, are represented by edges in which the flow control is indicated by the edge direction.

For most of the BPMN flow object elements, the flow-control graph representation is straightforward in that it can be represented by a graph vertex, with one or more input flows and with one or more output flows. For instance, the task and sub-process BPMN elements belong to the category of activities of BPMN symbols that are represented by graph nodes, each with one incoming and one outgoing flow object. A gateway can also be represented by a graph node with one or more incoming or outgoing edges. Events, which also belong to the category of activities, are not as obvious and shall be discussed here in more detail in terms of how they are represented in a DAG used for DE-HSM modeling.

There are three types of events: *Start, Intermediate,* and *End*. The Start and End events are special in that they occur at the very beginning and the very end of a process, respectively; hence, they are specialized cases of the flow of control. Events have a Dimension attribute that describes the type of an event, which may be one of message, timer, cancel, compensation, conditional, link, signal, terminate, multiple, and parallel. Out of this list, we currently support the following events: message, timer, conditional, and signal, as shown in Fig. 1. We are in the process of incorporating the support for the rest of the BPMN elements. We first discuss how intermediate events are represented and then discuss the representation of the begin and end events.

Intermediate events that are not boundary have explicit connections, represented by connecting objects between the corresponding throwing and catching events. Thus, they can be represented by vertices connected by edges that represent the connecting objects. A boundary event does not have an explicit connection via a connecting object to its counterpart, such as which signal in a task/sub-process is handled by which of the boundary catch events. An explicit connection between the rise of an event and its consequence, however, needs to be established, which is done at the conclusion of the modeling process when the modeler is asked by the transformation process to provide details on the flow of inputs and outputs along the edges of the graph. For each boundary catch event, the modeler is asked to identify, by using labels, which of the signal events caused the catch event by matching the labels of the throw event firing and it is catching it as a boundary event. Thus, each event does have at least one incoming and one outgoing edge. A boundary catch event, such as catching a signal event, is represented by a graph node that is connected to its corresponding throw signal event, within the activity to which it is attached (as a boundary event), by an incoming

edge from the vertex representing throwing the corresponding event. If the boundary activity is also interrupting, the state of execution for the interrupted activity is ended. However, if it is not interrupting, in essence, two concurrent activities will proceed, one to continue with the execution of the activity with the boundary condition, and one due to the catching of the boundary non-interrupting event and processing it.

A boundary *begin event* is a throwing event. If a boundary begin event is on a boundary of an activity, i.e., a sub-process or a task, that is, a graph node B, the boundary event is represented, on the DAG representation of the BPMN model, by inserting an inclusive gateway on the incoming edge to node B, as identified via label numbers between the boundary start event and the label of its corresponding catch event as provided by the modeler. In other words, boundary throwing events are fired/thrown at the beginning of an activity, and if it is non-interrupting, then two processes proceed concurrently during execution, one for the boundary start-event activity and one for the uninterrupted activity represented by vertex B to which the boundary start event is attached.

A boundary *end event* is a catching event, and it is represented in the DE-HSM model in the usual way by a graph node with the incoming edge that is also an outgoing edge from the corresponding throwing event, as represented by the labeling of events.

4.4. Transformation of a BPMN model into a DE-HSM multi-modal model

The DE-HSM model, which we are seeking and that represents a BPMN graph, is an interconnection of DE-HSM sub-models, where each sub-model is a child model for the parent model. Furthermore, each of the sub-models may itself be a DE-HSM sub-model, i.e., it may be modeled as an interconnection of its child models. As described in a previous subsection, the TABS system represents a BPMN model using a DAG, and then it determines the DE-HSM model as an interconnection of the sub-models, wherein each sub-model is represented without further sub-models: The topic of this section. However, once we do find the sub-models, we also need to describe how to build each sub-model, which is the topic of the next section.

Starting with a BPMN diagram, we first describe how we represent a BPMN model as a DAG and then how we approach defining the problem of finding a DE-HSM model for a given DAG. To find a DE-HSM model, we use the concept of independent subgraphs and their variations, which are described next. We then present an algorithm to find a DE-HSM model that is an interconnection of DE-FSM sub-models, i.e., sub-models that do not have hierarchical states.

4.4.1. DAG representation of a given BPMN graph

At execution time, a DE-FSM sub-model contains a DE queue of events and an FSM, and processing within a sub-model proceeds by dequeuing an event from the DE queue and then using it to form an input that is given to the FSM. Upon input, the FSM fires and produces output. The main property of the DE-FSM sub-model, however, is that it accepts inputs that are fed to an FSM, and outputs produced by the FMS firing produce the output out of the sub-model. Thus, a sub-model consumes one set of inputs and produces one set of outputs.

We represent a BPMN model as a DAG $G = (S, E)$, where S is the set of vertices and E is a set of edges. For a given BPMN diagram, vertices in S represent the BPMN non-connecting objects, while edges represent the connecting objects, that is, sequence flows or message flows. We use a DAG $G\hat{} = (S\hat{}, E\hat{})$ to represent a DE-HSM model that we find, i.e., a model in which:

1. For each element $S\hat{}_i \in S\hat{} = \{S\hat{}_i,\ i = 1, 2, \ldots n\hat{}\}$, $S\hat{}_i$ is a subset of vertices in S, $S\hat{}_i \subseteq S$.
2. For each element $e\hat{}_k \in E\hat{} = \{e\hat{}_k, k = 1,2, \ldots, m\hat{}\}$, $e\hat{}_k$ is an edge in E, such that each edge $e\hat{}_k = (s_x, s_y)$ corresponds to an edge $(s_x, s_y) \in E$, wherein $s_x \in S\hat{}_i$, $s_y \in S\hat{}_k$, $i \neq k$, $i = 1, 2, \ldots, n\hat{}$ and $k = 1, 2, \ldots, m\hat{}$
3. $S\hat{}_i \cap S\hat{}_k = \emptyset$, i.e., $S\hat{}_i$ and $S\hat{}_k$ are mutually exclusive.
4. For each $S\hat{}_i$, $i = 1, 2, \ldots, n\hat{}$, vertices in $S\hat{}_i$ are vertices of a subgraph $G'_i = \{S'_i, E'_i\}$, which is a subgraph of G, $G'_i \subseteq G$, such that $S'_i = S\hat{}_i$, i.e., vertices of $G\hat{}_i$ and G'_i are the same, and the set of edges E'_i includes all those edges $e = (s_x, s_y) \in E$, where $s_x, s_y \in S\hat{}_i$ and $S\hat{}_i = S'_i$; i.e., $s_x, s_y \in S\hat{}_i$ and the edges are internal to G'_i.
5. $(\cup S\hat{}_i,\ i = 1, 2, \ldots, n\hat{}) = S$: Union of all $S\hat{}_i$ is S, i.e., each vertex in S appears in exactly one of the $S\hat{}_i$.
6. The following properties apply to the sets of edges in E:
   - o $E\hat{} \cap (\cup E\hat{}_i, i = 1, 2, \ldots, m\hat{}) = \emptyset$; i.e., $E\hat{}$ intersect (union of all edges E'_i) is empty. Edges in $E\hat{}$ form interconnections amongst the subgraphs G'_i, where each subgraph G'_i represents a DE-FSM sub-model.
   - o Each edge $e \in E$ is either in $E\hat{}$ or in the union of all edges in E'_k, $k = 1, 2, \ldots, m\hat{}$.

In other words, each $S\hat{}_i$ is a set of vertices of a subgraph $G'_i = (S'_i, E'_i)$, where $S'_i = S\hat{}_i$ forms a set of vertices of a sub-model of the DE-HSM model, and edges in E'_i are all those edges $(s_x, s_y) \in E$ that are internal edges of the subgraph G'_i. Thus, each $S\hat{}_i \in S\hat{}$ identifies a DE-HSM sub-model and edges in $E\hat{}$ represent the interconnection of the DE-HSM sub-models. Furthermore:

- o G'_i must be a connected graph.
- o As a sub-model must have only one entry and one exit node to be represented by an HSM or an FSM, its subgraph can only have one entry and one exit vertex.

4.4.2. Finding a DE-HSM multi-modal model

Given a BPMN diagram, we represent it by a DAG $G = (S, E)$ for which we wish to find its equivalent DAG $G\hat{} = (S\hat{}, E\hat{})$ in which each node $S\hat{}_i$, $S\hat{}_i \in S\hat{}$, represents a subset of nodes of S that are members of a subgraph $G'_i = (S'_i, E'_i)$, such that:

1. Each G'_i represents a sub-model $G'_i = (S'_i, E'_i)$ of $G\hat{}$ and hence $G'_1, G'_2, \ldots, G\hat{}'_n$ are such that
   - o $G'_i \subset G$, $S'_i \subset S$
   - o $S'_i \cap S'_k$, $i \neq k$, i.e., the sets S'_i, $i = 1, 2, \ldots, n\hat{}$, are mutually exclusive
   - o $S = \cup S'_i$, $i = 1, 2, \ldots, n\hat{}$; vertices in subgraphs cover the whole graph S
   - o For each edge $e = (s_x, s_y) \in E$:
     - $e \in E\hat{}$, i.e., it is one of the interconnections of the sub-models, or
     - $e \in E'_i$, for some i and where $E'_i \cap E\hat{} = \emptyset$, $i = 1, 2, \ldots, n\hat{}$, i.e., intersection of the set of edges that interconnects the sub-models with any of edges of the subgraphs is empty
2. $G'_1, G'_2, \ldots, G'_n$ are such that for G'_i and G'_j, where G'_i and G'_j are both subgraphs G', are either
   - o disconnected, i.e., not connected to each other directly (but connected indirectly via transitivity through some other $G_k \in G'$), or are
   - o connected by one edge $e = (s_x, s_y)$, where $s_x \in S_i'$ and $s_y \in S'_k$, $i \neq k$
3. Each sub-model has exactly one entry point and exit point with the following two exceptions:
   - o The subgraph that contains the Start event node of the original BPMN graph does not have any entry node, as there is no incoming edge from the subgraph's external nodes to any of the subgraph's internal nodes.
   - o The subgraph that contains the End event node of the original BPMN graph does not have any exit node, as there is no outgoing edge from any of the subgraph's nodes to any of the subgraph's external nodes.

The conditions in (i) ensure that the subgraphs are mutually exclusive and that they cover the whole BPMN graph. Condition (ii) states that each subgraph is connected to some other subgraph by exactly one edge. The last condition, condition (iii), is due to the fact that each DE-HSM sub-model accepts input fed to its FSM that produces output upon

firing. Or rephrasing it, a sub-model has an FSM, and the sub-model's input forms the input to its FSM. After the FSM fires, its output forms the sub-model's output. As a consequence, each subgraph G_i', which represents a sub-model in $G^{\wedge}$, must be such that it has one vertex, the subgraph's entry node/vertex, that receives the FSM's input, and one node/vertex, the subgraph's exit point, that forwards the FSM's output for further processing after exiting out of the subgraph.

4.4.3. Independent subgraphs

We now present the concept of *independent* subgraphs.

Definition. Let G be a graph, $G = (S, E)$. A subgraph $G' = (S', E')$, $G' \subset G$, is called an independent subgraph if it satisfies the following properties:

1. All nodes of the subgraph G', except for its entry and exit nodes, have only internal edges (x, y), where $x, y \in S'$.
2. There exists exactly one node/vertex, s_{en}, in the subgraph such that, in addition to its internal edges, it has only one vertex with incoming edges from external nodes; such a node is referred to as an entry node. (An entry node does not have any outgoing edges to external nodes.) The exception is that node of G' that represents the BPMN Start event element as it does not have any incoming edges from external nodes.
3. There exists exactly one node/vertex, s_{ex}, in the subgraph that, in addition to its internal edges, has any outgoing edges to any external nodes; such a node is referred to as an exit node. The exception is the node of G that represents the BPMN END event element as it does not have any outgoing edges to external nodes.

In the above definition, except for the entry and exit nodes, all internal nodes of an independent subgraph have edges connected only to other internal nodes. As graph S represents the control flow and subgraph S' has only one entry and one exit node, this means that once the initial state of the subgraph is reached during execution, the execution will proceed locally within subgraph S' until execution reaches the subgraph's exit vertex, s_{ex}. That is exactly the property that is required for a DE-FSM sub-model. We now use Theorem 1, presented in Refs. [40,41], to show the properties of independent subgraphs.

Theorem 1. *Consider a BPMN model, which is represented as a DAG $G = (S, E)$, and that has two independent subgraphs: $G_1 = (S_1, E_1)$ and $G_2 = (S_2, E_2)$, where $S_1 \subset S$ and $S_2 \subset S$ and where $|S_1| > 1$ and $|S_2| > 1$. Then, one of the following holds:*

a) *$S_1 \subset S_2$ or $S_2 \subset S_1$*
b) *S_1 and S_2 are such that they either*

(i) *share nodes that form an independent subgraph or*
(ii) *they share at most one edge, and furthermore, that edge is such that it is an outgoing edge from an exit node of one of the subgraphs, and furthermore, that edge is also an incoming edge to the entry node of the other subgraph.*

Proof: The proof can be found on pages 39–41 of Ref. [42].

We will use the concept of an independent subgraph to determine the DE-HSM model. However, a DE-HSM model may be such that it contains a number of other DE-HSM models that are also independent subgraphs. We are searching for independent subgraphs such that each one is represented by the DE-FSM sub-model, that is, a subgraph that does not have any further internal sub-models. However, we do not wish to decompose unnecessarily.

Definition. Let $G = (S, E)$ be a DAG. A subgraph G', $G' \subset G$, is called a Smallest Independent (SI) subgraph if it has more than one vertex and does not have any proper subset that is also an independent subgraph.

We can use SI subgraphs to form the sub-models, however, such a decomposition breaks up sub-models when there is no need for further decompositions. Consider Fig. 5, which shows an independent subgraph

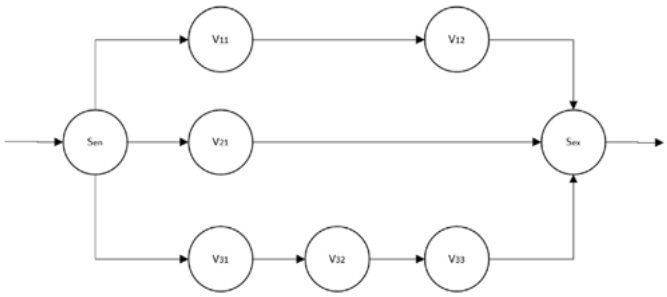


Fig. 5. Independent subgraph with proper subsets that are also independent subgraphs.

G', of a part of a BPMN graph of $G = (S, E)$. $G' = (S', E')$, where $S' = \{ s_{en}, v_{11}, v_{12}, v_{21}, v_{31}, v_{32}, v_{33}, s_{ex} \}$, $E' = \{ (s_{en}, v_{11}), (s_{en}, v_{11}), (v_{11}, v_{12}), (v_{12}, s_{ex}), (s_{en}, v_{21}), (v_{21}, s_{ex}), (s_{en}, v_{31}), (v_{31}, v_{32}), (v_{32}, v_{33}), (v_{33}, s_{ex})\}$. However, G' has two independent subgraphs: $G_1 = (S_1, E_1)$ and $G_2 = (S_2, E_2)$, where.

1. $G_1 = \{v_{11}, v_{12}, v_{21}, s_{ex}\}$ and $E_1 = \{(s_{en}, v_{11}), (v_{11}, v_{12}), (v_{12}, s_{ex}), (s_{en}, v_{21}), (v_{21}, s_{ex})\}$
2. $G_2 = \{v_{21}, v_{31}, v_{32}, v_{33}, s_{ex}\}$ and $E_2 = \{(s_{en}, v_{21}), (v_{21}, s_{ex}), (s_{en}, v_{31}), (v_{31}, v_{32}), (v_{32}, v_{33}), (v_{33}, s_{ex})\}$

Clearly, it is desirable to have only one DE-HSM model representing the subgraph G' as opposed to the two sub-models, one for G_1 and one for G_2. Further decomposition of G' into G_1 and G_2 is not only unnecessary, but it would also result in a more complex interconnection of the sub-models than is necessary. Another example is an independent subgraph $G'' = (\{v_{31}, v_{32}\}, \{(v_{31}, v_{32})\})$ or an independent subgraph $G''' = (\{v_{31}, v_{32}, v_{33}\}, \{(v_{31}, v_{32}), (v_{32}, v_{33})\})$. They are both independent subgraphs, but clearly, there is no need to have a separate sub-model to represent them.

We note that for any smallest independent subgraph $G' = (S', E')$, any outgoing edge from the entry vertex, s_{en}, identifies a path of connected nodes such that the path terminates at the exit node, s_{ex}. Furthermore, any node on that path has only two edges that are on the path from s_{en} to s_{ex} and no other edges. If a node, say a node $s_1 \in S'$, which is on a path from s_{en} to s_{ex}, did have another edge, say to a node $s_2 \in S'$, that is not on the same path from s_{en} to s_{ex}, it would violate the properties of the smallest independent subgraphs as either: (i) If $s_2 \notin G'$, then an edge from the node s_1 to s_2 within the subgraph G' leads to the node s_2 that is outside the subgraph G', which means that the subgraph is not an independent subgraph as it has two nodes with edges outgoing to nodes that are external to the subgraph. Or (ii) $s_2 \in G'$, but then the edge (s_1, s_2) would create an independent subgraph within the smallest independent subgraph G', which is a contradiction.

4.4.4. Largest Smallest Independent (LSI) subgraph

Consequently, we wish sub-models to represent independent subgraphs such that each subgraph does not have any independent subgraph itself unless that the independent subgraph (i) has the same entry and exit nodes as its parent graph or (ii) is fully on a path from the subgraph's entry node to its exit node. We introduce the concept of the Largest Smallest Independent (LSI) subgraph.

Definition. Let G be a graph, $G = (S, E)$. A subgraph $G' = (S', E')$, $G' \subset G$, is called the Largest Smallest Independent (LSI) subgraph of G if it is an independent subgraph, and furthermore, any vertex $s \in S'$, which is neither an entry nor an exit node of an independent subgraph, has exactly one incoming and one outgoing edge.

We will use the properties of the LSI subgraphs to construct, from the given graph G, an equivalent representation $G^{\wedge} = (S^{\wedge}, E^{\wedge})$, where each $S^{\wedge}_i \in S^{\wedge}$ is a set of vertices of a subgraph G'_i, $i = 1, 2, \ldots, n^{\wedge}$. Each subgraph G'_i is an LSI subgraph that represents a DE-FSM sub-model, and $E^{\wedge}$ is a set of

edges in E that represents interconnections of the sub-models represented by the subgraphs.

4.4.5. Algorithm to find LSI subgraphs

The previous sections dealt with formalities and abstractions suitable for the presentation of the concepts of independent and LSI subgraphs. We now provide a pseudocode of an algorithm to find LSI subgraphs for a given DAG G that represents a BPMN model. The algorithm, called *findLSIsubgraphs,* has as its input a global DAG $G = (S, E)$, which represents a BPMN diagram. The method results in a DAG $G^{\wedge} = (S^{\wedge}, E^{\wedge})$, such that $E^{\wedge}$ represents edges and each $S^{\wedge}_i \in S^{\wedge}$ represents a subset of nodes S'_i of the graph G'_i, where the nodes of $S_i^{\wedge}$, $i = 1, 2, \ldots, n^{\wedge}$, form a subgraph $G_i' = (S'_i, E'_i)$, where $S'_i = S^{\wedge}_i$ is a set of nodes representing an LSI subgraph that is an FSM sub-model. Furthermore, E_i' represents interconnections of the nodes in S'_i, wherein each edge $e = (s_x, s_y)$ is an internal edge to S'_i, i.e., where $s_x \in S'_i$ and $s_y \in S'_i$. Thus, each $S^{\wedge}_i$ defines an LSI subgraph $G' = (S'_i, E'_i)$, while the set of edges $E^{\wedge}$ of $G^{\wedge}$ form interconnections amongst the LSI subgraphs.

The method uses a global variable $G = (S, E)$ to mark the nodes with information on vertices' incoming and outgoing degrees, where the outgoing degree of a vertex s is the number of directed edges of the form (s, s_x), for some vertex $s_x \in G$, and an incoming degree is the number of edges of the form (s_x, s), where $s_x \in G$.

Recall from Section 3.1.1, which dealt with the preprocessing of the BPMN graph, that the preprocessing replaces any gate that is both a join gate and a fork gate with an equivalent representation by one merge gate and one fork gate, wherein the merge gate has one or more incoming edges but only one outgoing edge that is also the only incoming edge to a fork gate that has more than one outgoing edge.

Before we proceed with the presentation of the algorithm to find LSI subgraphs, we make a few observations about LSI subgraphs:

a) With the exception of the entry or exit nodes, there is no vertex that has both incoming and outgoing degrees greater than one.
b) If a vertex has an incoming degree of more than one, then it is a merge/join gate with one outgoing edge. In such a case, the vertex may only be an exit node from an LSI subgraph.
c) If a vertex has an outgoing degree of more than one, then it is a fork/split gate with an incoming degree of one. In such a case, the vertex may only be an entry node to an LSI subgraph.
d) An internal node of an LSI subgraph has an incoming degree and an outgoing degree that are exactly one.

The algorithm/method *findLSIsubgraphs*, shown in Fig. 6, has a DAG G

```
(1)  Input: DAG G = (S, E);
(2)  Output: setOfLSI subgraphs G* = { Gi' = (Si',Ei'), i = 1, 2, … }
(3)              where each Si' corresponds to the Si^ an element of S^, where G^ = (S^, E^),
(4)              and E^ is the interconnection of the LSI subgraphs Gi', i = 1, 2. …
(5)  BEGIN
(6)  Initialization;
(7)  for each node s in S do
(8)     Let s.inDegree = the number of incoming edges to s ;
(9)     Let s.outDegree = the number of outgoing edges from s ;
(10)    Let s.root = s ;
(11) end
(12) Let G* be empty;
(13) var vertex s, sx, sy;  var edge e = {sx, sy};   var LSIsubgraph G' = (S', E') ;
(14)
(15) // invocation of the breadth-first search traversal of the DAG G ;
(16) while ( ( s = nextInLevelOrder (G)) ≠ nil ) do
(17)    // Check if vertex s represents the START BPMN element
(18)    if (s.inDegree = 0) then
(19)       let s.root = s  ;  //Assumption: BMPN diagram has more than the START and END element
(20)       exit while loop;
(21)    end
(22)    // s may be an internal LSI subgraph node or an exit node or may represent END element;
(23)    if (s.inDegree > 1)  then   // s node is an exit node from an LSI subgraph
(24)       G'i = isLSIsubgraph (s.root, s);  // Should return an LSI subgraph
(25)       add G'i to G*;   // G'i should not be null
(26)       exit while loop;  // exit the current iteration of the while loop and check for the next iteration
(27)    if (s.inDegree = 1)  then
(28)       if (s.outDegree = 0)  then //s represents the END BPMN element => it is exit node
(29)          G'i = isLSIsubgraph (s.root, s);  // Should return an LSI subgraph
(30)          add G'i to G*;                 // G'i should not be null
(31)          exit while loop;  // exit the current iteration of the while loop and check for the next iteration
(32)       elseif (s.outDegree = 1) then    // there is only one edge from s and s may or may not be an exit node
(33)          Let e =( s, sy);  // Find the destination vertex sy of the only outgoing edge from s
(34)          if ( sy.inDegree > 1) ) then  //  s is an exit node from an LSI subgraph as sy is an entry node
(35)             G'i = isLSIsubgraph (s.root, s);  // Should return an LSI subgraph
(36)             add G'i to G*;   // G'i should not be null
(37)             exit while loop;  // exit the current iteration of the while loop and check for the next iteration
(38)          else
(39)             if ( sy.inDegree = 1) then  // s is neither exit nor entry node
(40)                s.root = (pointedToBy(s)).root=s ;
(41)             endif  // if ( sy.inDegree = 0) then  * The node sy represents the END BPMN that will be
(42)                                                    recognized as such and processed on the next iteration */
(43)          endif
(44)       endif
(45)    endif
(46) endwhile;
(47)  // Post processing:
(48) Combine any subgraphs G' in G* that have the same entry and exit nodes into one subgraph;
(49) END;
```

Fig. 6. Algorithm *findLSIsubgraphs()*.

$= (S, E)$ as its input. Recall that the Start event has only one edge, which is an outgoing edge. Each node $s \in S$ is assumed to have the following attributes:

- ***s.inDegree*** … the number of edges that are incoming into the vertex *s*
- ***s.outDegree*** … the number of edges outgoing out of the vertex *s*
- ***s.root*** … reference to a node that may be an entry of an LSI subgraph to which the node *s* may belong

The method *findLSIsubgraphs()* uses the following auxiliary methods:

- *vertex x = nextInSearchOrder (G)* … the method facilitates a breadth-first search traversal of the connected DAG *G*, in which that vertex $s \in S$ that represents the BPMN Start event is the start of the traversal. Each of the other nodes has at least one incoming edge and one or more outgoing edges, with the exception of the vertex that represents the BPMN End event, which does not have any outgoing edges. Calling the method *nextInSearchOrder(G)* repeatedly results in the method returning a reference to the next node in the breadth-first search traversal of *G* and returning null once all vertices have been traversed.
- *LSIsubgraph* G'_i *= isLSIsubgraph (s, d)* … given vertices *s* and *d*, the method determines whether the vertices *s* and *d* identify an LSI subgraph by checking the cases that are discussed below. If it does discover an LSI subgraph, it is returned by the method; otherwise, the method returns null. We first discuss the special cases that exist for the vertices that represent the BPMN Start and End event elements.
  - o If the in-degree of the vertex, say *s*, is zero, then the vertex represents the BPMN Start element, and hence cannot be an exit node. The s.root attribute is set to s as it will be the entry node of an LSI subgraph, which is also the root node of the subgraph.
  - o If the out-degree of the vertex is zero, then the vertex represents the BPMN End event element. It is treated as an exit vertex of an LSI subgraph.
- *vertex* s_x *= pointedToBy(s)* … The method returns null if the vertex *s* does not have any incoming edges or if it has more than one incoming edge. Otherwise, it returns that vertex s_x that is the source for the only edge leading to *s*, i.e., it returns that vertex s_x for which there exists an edge $e = (s_x, s)$.

The algorithm uses a breadth-first search traversal of the DAG *G*, starting with graph node *s* that represents the BPMN Start event element, while its output has been described already. We now discuss the details of the *findLSIsubgraphs()* algorithm that is shown in Fig. 6.

The first part of the algorithm deals with initialization that sets the attributes of each node of the DAG *G*, namely, s.inDegree, s.outDegree, and s.root. Following that, there is a while loop that is used to determine whether the node *s*, returned by the *nextInSearchOrder (G)*, is an exit node of an LSI subgraph and whether that node identifies an LSI subgraph, which has been found and should be included in the set of LSI subgraphs output by the method, or whether it is some other case and how it should be handled. These cases are now discussed by making specific references to the lines of the algorithm presented in Fig. 6.

Line 18 within the while loop checks the in-degree of the vertex *s*. If the in-degree of the vertex *s* is zero, then the vertex *s* represents the BPMN Start element, and hence it is an entry node of an LSI subgraph. If it is, then the s.root attribute is set to *s* as it will become a root node of an LSI subgraph. The while iteration exits and another iteration starts (lines 24–26). Note that for the vertex representing the BPMN Start element, its out-degree cannot be greater than one, as only a fork gate can have an out-degree greater than one.

- If the s.inDegree >1 (line 23), then the vertex *s* is an exit node, an LSI subgraph has been discovered and added to the set of LSI subgraphs $G^{\wedge}$, and the loop is exited (lines 24–26).
- If the s.inDegree is equal to 1 (line 27), then the node's out-degree is checked.
  - o If the out-degree of the vertex *s* is zero (line 28), then the vertex *s* represents the BPMN End node, and consequently, the LSI subgraph identified by the entry and exit nodes s.root and *s*, respectively, is returned as an LSI subgraph G' that will be added to the set of LSI subgraphs $G^{\wedge}$ (lines 29–31).
  - o If the out-degree of the vertex *s* is equal to one (line 32), then it cannot be determined as yet whether the node *s* is or is not an exit node of an LSI subgraph. The vertex *s* could be an exit node such that there is only a single path to the node *s*, while the outgoing edge from *s*, edge (s, s_y), leads to a node that has an out-degree greater than one and hence be the entry node of another LSI subgraph, in which case the node *s* is an exit node from an LSI subgraph. If the out-degree of the node s_y is one, then the node *s* is not an exit node from an LSI subgraph, as the node s_y could be safely added to the LSI subgraph identified by the vertices s.root and s without violating the LSI subgraph properties. Thus, to determine whether or not *s* is an exit node, we must examine the out-degree of the node s_y, where s_y is the destination node of the outgoing edge from *s* (line 33):
    - If the out-degree of the vertex s_y is greater than zero (line 34), then the vertex s_y, where s_y is the destination vertex of the outgoing edge from *s*, i.e., the edge (s, s_y), is the entry node of the next LSI subgraph, and hence, the node *s* is the exit node of the LSI subgraph. The found LSI subgraph is added to the set of LSI subgraphs that are output (lines 35–37).
    - If the out-degree of the vertex s_y is equal to 1 (line 39), then the vertex *s* is neither an exit nor an entry node, and the s.root attribute is set to refer to the s.root attribute of the vertex that has an outgoing edge that points to *s*.
    - As a note, if the out-degree of the vertex s_y is equal to 0, then s_y represents the BPMN End element, in which case nothing needs to be done as the node s_y will be processed on a subsequent iteration of the while loop.

Once all nodes are processed, the algorithm performs a post-processing step that ensures that any LSI subgraphs that were discovered and added to the set $G^{\wedge}$ and that have the same entry and exit nodes are combined into one LSI subgraph. For instance, if the algorithm outputs two separate independent subgraphs G_1 and G_2, where G_1 and G_2 are:

1. $G_1 = \{v_{11}, v_{12}, v_{21}, s_{ex}\}$ and $E_1 = \{(s_{en}, v_{11}), (v_{11}, v_{12}), (v_{12}, s_{ex}), (s_{en}, v_{21}), (v_{21}, s_{ex})\}$
2. $G_2 = \{v_{21}, v_{31}, v_{32}, v_{33}, s_{ex}\}$ and $E_2 = \{(s_{en}, v_{21}), (v_{21}, s_{ex}), (s_{en}, v_{31}), (v_{31}, v_{32}), (v_{32}, v_{33}), (v_{33}, s_{ex})\}$

Then the post-processing step combines all independent subgraphs that have the same entry and exit nodes into one subgraph. For the example of G_1 and G_2, the post-processing step combines the two subgraphs into one, as shown in Fig. 5.

The significance of decomposing the BPMN model into DE-HSM sub-models that satisfy the property of LSI subgraphs is that the LSI subgraphs lead to sub-models that can be modeled directly by a DE-FSM model, i.e., they do not need any further decomposition into sub-models.

4.4.6. Example

Consider the sample supply chain BPMN model shown in Fig. 3 that was also used in Refs. [16–18]. Applying the algorithm *findLSIsubgraphs()* results in the LSI subgraphs, listed by the method *foundSubgraph*, which are shown in Fig. 7 and are shown below.

- S_1 This subgraph contains INIT, buyer send offer, manufacturer receive order, calculate demand, manufacturer place order, and middleman receive order.

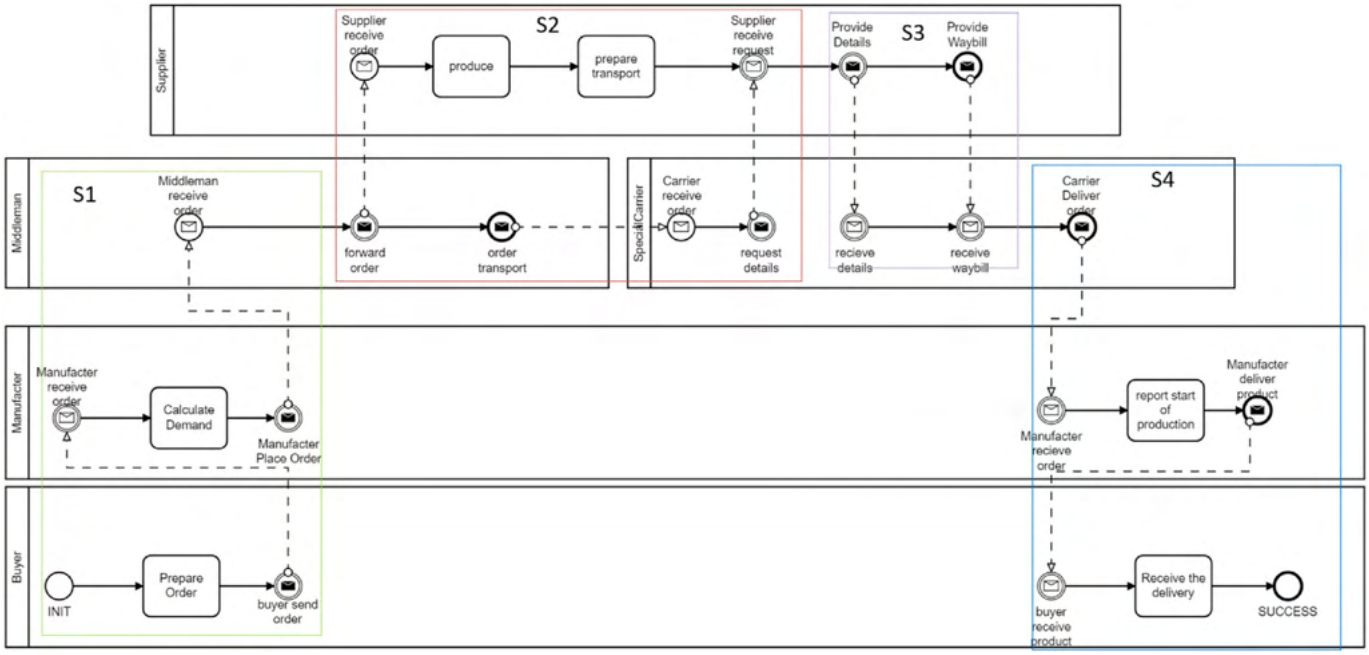


Fig. 7. BPMN model and found subgraphs.

- S_2 This subgraph contains forward order, order transport, supplier receive order, produce, order transport, prepare transport, carrier receive order, supplier receive request, and request details.
- S_3 This subgraph contains provide details, provide waybill, receive details, and receive waybill.
- S_4 This subgraph contains carrier deliver order, manufacture receive order, report start of production, manufacturer deliver product, buyer receive product, and SUCCESS.

To explore how the algorithm works, consider its execution after its initialization, when the while loop is executed. The breadth-first search traversal returns the INIT node as the first node. Thus, the node s in the while loop refers to the INIT node, which represents the BPMN Start element, on the first iteration of the loop, at which point s.root = s, which represents the fact that the node s will be the entry node of the LSI subgraph that will be discovered. Following the first loop, subsequent loops retrieve nodes: Prepare Order, buyer send order, manufacturer receive order, and Calculate Demand. All the listed vertices have exactly one incoming and one outgoing edge and thus cannot be exit nodes of an LSI subgraph. As a result, their processing by the while statement will set the attribute s.root to refer to the vertex representing the INIT node, as set by line 39; that the vertex will become a root of an LSI subgraph.

Assume that all nodes from the node INIT to Calculate Demand have been processed by the while loop, and now the reference to the node Manufacturer Place Order has been returned by the method in the while statement *while* s = ((*nextInSearchOrder (G)*) $\neq$ *nil*) *do,* and that the reference to that vertex has been stored in the variable *s.* Since both the in-degree and out-degree are one for the node s, lines 33–40 of the method in Fig. 6 are executed next. As the out-degree of the node s_y, where the vertex s_y is the destination vertex of the edge (s, s_y), is also 1, the vertex s is neither an entry node nor an exit node, and hence the attribute s.root is set to s.root = (pointedToBy(s)).root = s, as was done by while loop processing of the previous vertices.

In the next iteration of the while loop, the node returned by the breadth-first search method is the node s that represents the Middleman receive order BPMN element. As both the in and out degrees are equal to 1, lines 33–40 are executed. As the out-degree of the node s_y, where the vertex s_y (represents the forward order BPMN element) is the destination vertex of the edge (s, s_y), is greater than 1, then the vertex s_y is the entry node of the next LSI subgraph, and hence the node s is the exit node of the subgraph. Consequently, the statements in lines 35–37 are executed and an LSI subgraph, with the entry node being the vertex representing the INIT element and the exit node being the Middleman receive order BPMN element.

4.5. Building DE-FSM sub-models and their interconnection

Thus far, we have described how to determine a DE-HSM model, that is, how to find a DE-HSM model for which each of its sub-models does not contain any further sub-models. In this section, we describe how to build each of the DE-FSM sub-models and interconnect them.

4.5.1. Building a DE-FSM submodel

The previous section described how to find a DE-HSM model for a given graph G that represents the BPMN model. The DE-HSM model is represented by a graph $G\hat{} = (S\hat{}, E\hat{})$, such that each $S\hat{}_i \subset S\hat{}$ represents a subset of nodes of the graph G, where the nodes of $S\hat{}_i$, $i = 1, 2, \ldots, n\hat{}$, are also the nodes of a subgraph $G'_i = (S'_i, E'_i)$ of G, where $S'_i = S\hat{}_i$ is a set of nodes representing an LSI subgraph that is an FSM sub-model.

Given an LSI subgraph $G' = (S', E')$ that represents a DE-FSM sub-model, we build its model using a breadth-first search traversal of its graph $G'_i = (S'_i, E'_i)$, starting at its entry node s_{en}. Recall that each sub-model represents a DE-FSM model that uses one DE queue, which is shared by all sub-models, and an FSM that represents functionality. Also recall from the previous section that described how to find the DE-HSM model that a fork gate denotes an entry node of an LSI subgraph, while a merge gate is an exit node/vertex from an LSI subgraph. However, an LSI subgraph is such that each of its vertices, that is, neither the entry nor the exit vertex of the subgraph has exactly one incoming and one outgoing edge. Thus, with the exception of its entry or exit nodes, there are no other vertices that represent a gate. As a consequence, the translator creates the sub-model in two phases.

The first phase corresponds to the flow control entering the sub-model during the execution phase; that is, when the flow control reaches the subgraph's entry node. Depending on the type of the entry node, further execution may proceed as one processing stream for an exclusive fork gate or more than one processing stream in the case of an inclusive fork gate. Note that an exclusive or inclusive fork gate may have a guard

along each outgoing edge, wherein a guard along an outgoing edge represents the constraint that the FSM's input must satisfy for the execution to proceed along that edge. Of course, for an exclusive gate, guard constraints must be such that only one evaluates to true at any one time. Thus, an inclusive fork gate also covers a parallel fork gate when all guards are set to true.

If the entry node of the LSI subgraph represents an exclusive fork gate BPMN element, then in the first phase, the root of the FSM is created. For an exclusive fork gate, in addition to the root of the FSM, additional states are appended for a binary search that will result in identifying a path that represents the control flow along each of the paths from the entry node to the exit node. For instance, if there are four outgoing edges from an exclusive fork gate with the guards being A, B, C, and D, then Fig. 8 shows the vertices added to the FSM in order to identify the path for each of the outgoing edges to the exit node. In the second phase, in each iteration of the search, a new state is added on a path for which the input satisfies the guards. In short, there is one FMS for an entry node that represents an exclusive fork gate, such that the FSM's root is followed by transitions that result in n paths, one path for each of the outgoing streams from the gate. In the subsequent phase, state transitions are built along the outgoing paths, each one representing functionality for one of the guards on the exclusive fork gate.

If the entry node represents an inclusive fork gate BPMN element, then the translator creates n+1 FSMs if there are n outgoing streams out of the inclusive fork gate. In addition, one FSM, referred to as a control FSM, is created to determine at the execution time, to which of the outgoing streams from the fork gate does the input to the FSM belongs when an FSM fires, i.e., when it receives input to which it needs to react. The controlling FSM is used in the execution phase to determine to which of the FSMs should the input be directed depending on the evaluation of the guards; the identification of the FSMs for which guards evaluate to true is the output of the controlling FSM's firing. In addition, when the exit from the firings of the controlling FSM is reached, the flow of control is directed towards each of the concurrent FSMs for which the guard is evaluated to be true. This is achieved by the monitor component at runtime by creating a DE event for each of the guards that evaluates to true. For each one, the monitor inserts a DE event that causes it to fire and hence causes execution of the firing of the corresponding concurrent FSM when the event reaches the head of the queue. Fig. 9 shows the inclusive fork gate and the resulting control FSM and the concurrent FSM.

Once the FSMs are prepared, a breadth-first search of the nodes of the subgraphs is used to build the transition states of the concurrent FSMs. For an exclusive fork gate, all transitions are in the one FSM along the paths from the root to the exit node, wherein the initial vertices are created by the translator in the initial phase to guide the input to the appropriate path using the guard conditions. For an inclusive fork gate, for each node of the breadth-first search, the translator first determines to which of the FSMs the node belongs, and then the translator augments the corresponding FSM with the state transition for the node. Both phases are now described.

a) *Initial phase—Entering a sub-model*

The translator examines the BPMN element that the entry vertex of the sub-model represents, and the transformation into a sub-model depends on whether the BPMN element is not a gate, is an exclusive fork gate, or is an inclusive fork gate:

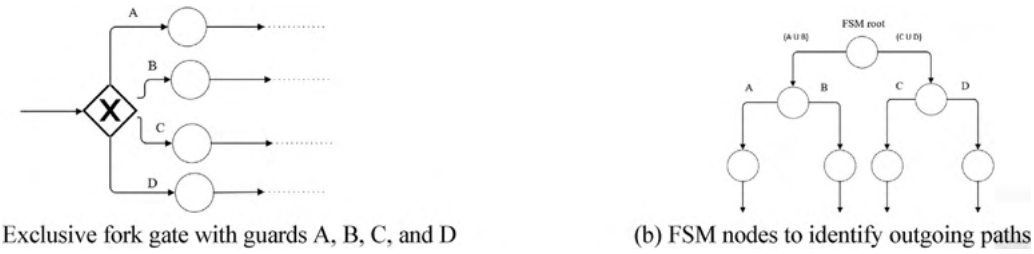
- **The entry node is not a gate.** The processing within the sub-model is represented by a single stream with the functionality represented by an FSM. The translator builds the root of the FSM, with the rest of the FSM states created by examining each of the nodes of the subgraph as described below in the main phase that builds the FSM transitions.
- **The entry node is an exclusive fork gate.** The processing within the sub-model is represented by a single stream with the functionality represented by one FSM. The translator builds the root of the FSM, and then it inserts states that check the input and perform a binary search using guards of the exclusive gateway to determine on which path, from the entry node to the root, is a new transition inserted (as shown in Fig. 8).
- **The entry node is an inclusive fork gateway.** The processing within the sub-model is represented by multiple concurrent streams by creating one concurrent FSM for each of the outgoing concurrent streams outgoing from the entry node that represents the inclusive fork gate BPMN element. In addition, the translator also generates the control FSM that is used to direct input coming into the sub-model into an appropriate FSM for an outgoing flow-control stream determined by examining the input to the FSM using constraints represented by the guards on outgoing edges of the BPMN gateway. As in the previous case of an exclusive gateway, the translator inserts into the controlling FSM states that check the input using Boolean expressions appearing in the guards of the gate that the node represents, as shown in Fig. 9.

b) *Main phase*

In the main phase, subgraph vertices are processed, vertices that have exactly one incoming and one outgoing edge. Such vertex results in a new FSM state and transition to be added to an FSM. For an inclusive fork gateway, the new state, and its transition are added to the appropriate concurrent FSM, while for the exclusive fork gateway, the new state and its transition are added to the appropriate path of the one FSM that has one path, from the subgraph's entry node to its exit node, for each of the outgoing edges out of an inclusive fork gateway.

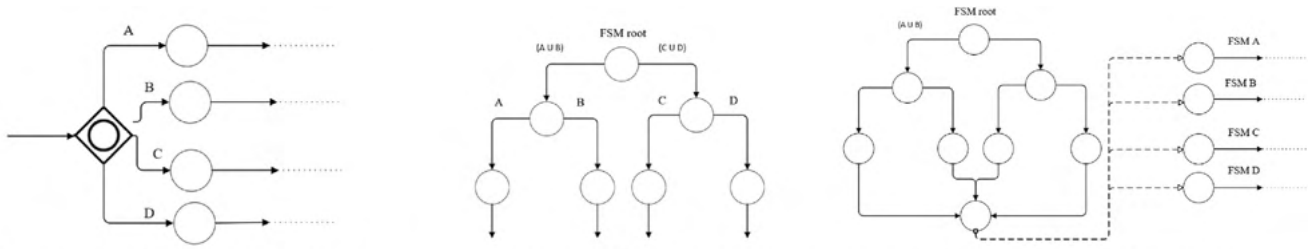
4.5.2. *Connecting the sub-models*

Once the sub-models are built, the overall model is built by finding the subgraph that contains the node that represents the BPMN Start event—call it the start subgraph. The overall model is now built by interconnecting the sub-models. The exit node of the start subgraph is connected to the entry nodes of LSI subgraph i, for which there is an outgoing edge from the exit node of the start subgraph that leads to the entry node of the LSI subgraph i. For each such interconnection, the translator takes the output from the FSM of the exit node and creates a DE event that is queued into the DE queue, with the event being targeted for processing by the FSM that represents the functionality of the LSI subgraph i. This process repeats for each of the LSI subgraphs.



(a) Exclusive fork gate with guards A, B, C, and D (b) FSM nodes to identify outgoing paths

Fig. 8. Entry node that represents an exclusive fork gate and FSM transitions as paths of FSM.



(a) Inclusive fork gate with guards A, B, C, and D

b) Controlling FSM and roots of n concurrent FSMs, one for each concurrent stream

Fig. 9. Entry node that represents an inclusive fork gate and control FSM, and one FSM for each of the concurrent process streams.

5. Transforming a DE-HSM model into methods of a smart contract

In the following subsections, we overview the transformation of the DE-HSM model into the methods of a smart contract. We only perform an overview, as our approach has been described in Refs. [32,43,44] and, in some respects, is similar to the work of others. However, we differ in providing the developer with the functionality to deploy parts of the smart contract to be processed on a sidechain for the purposes of reducing the cost and/or increasing privacy, and those are the aspects that we highlight in more detail.

5.1. Creating and deploying smart contract methods

During the contract execution phase, user participants and their systems interact with a DApp that, in turn, makes calls to the DApp API that our system provides. At run-time, once a smart contract method is invoked, its input is analyzed, and it is used to create an appropriate event that is queued into the DE queue. The DE queue contains events that are continuously dequeued and processed. Processing of a DE event includes (i) analysis of the DE event that determine which FSM is to fire and prepares input for that FSM; (ii) input is given to the FSM; (iii) FSM fires and produces output; and (iv) FSM output is transformed into a response to the smart contract method invocation. This is all performed under the control of the monitor/mediator, using information prepared during the design phase.

Thus, the run-time architecture of the system is similar to other approaches in the way the system interacts with the DApp and performs mapping of the DApp interactions with the API into calls to the methods of smart contracts and how the system responds to DApp requests. Where we depart is in how the flow control is managed.

The internal representation of the state of the system is based on the DE-HSM model that determines the flowcontrol between the FSM sub-models that represent the state of computation of individual DE-FSM sub-models. We faithfully model the BPMN graph using a DE-HSM model that is used to channel input into the appropriate sub-model, where the input is provided to the sub-model's FSM that fires and produces output. Depending on the DE-HSM model, the output produced by the FSM may be used as input to another sub-model and the process repeats, or the output is analyzed and provided as output for the call to the smart contract method, which in turn provides its returned parameters to the DApp method that invoked the DApp API. The channeling of outputs produced by sub-models is according to the interaction between the sub-models as defined by the DE-HSM model, while the function of each sub-model is represented by an FSM, or a group of FSMs if the sub-model has concurrent processing indicated by the inclusive fork gates, which also faithfully models a part of the BPMN graph that is a BPMN LSI subgraph.

We also depart from the work of others in that we facilitate off-chain processing on a sidechain, which is described next.

5.2. Sidechain processing for privacy and cost reduction

After the BPMN model is transformed into a DE-HSM model and individual sub-models are developed, but before the transformation into the methods of a smart contract, the developer is offered a design choice to deploy a part of the smart contract for processing on a sidechain for the purposes of privacy, reducing the processing cost, or both. To make that decision, sub-models are presented to the developer to select those that should be deployed on a sidechain. Actually, as a part of testing and before the final deployment, two versions of the smart contract are developed and deployed: In one, the sub-models selected by the developer are deployed and executed on a sidechain, while in the second version, all smart-contract methods are deployed and processed on the mainchain. Delays are measured for both versions, and the results are provided to the developer so that she/he can compare the cost of the smart contract execution, on the mainchain-only versus on the mainchain with sidechain processing of sub-models selected by the developer. The developer has information on the incurred delays in both cases, and if the mainchain or sidechains use Ethereum EVM, such as Quorum does, cost estimates in ETH gas are also provided to the developer.

We first describe our system architecture for the run-time phase with sidechain processing. We then describe how we assist the developer in determining which parts of the smart construct are suitable for sidechain processing, either for privacy or for reducing cost. Examples of the use of sidechains are provided in the following section that describes our PoC, the TABS tool.

5.2.1. System architecture with sidechain processing

Fig. 10 shows the system architecture at run-time for the mainchain and for the sidechain. The internal architecture for sidechain processing is in large part similar to that for the mainchain, which is shown in Fig. 2. Interoperability between the mainchain and the sidechain is achieved in part by a mainchain contract method calling a smart contract method on a sidechain. There are, however, additional issues involved, as our sidechain contract is really an integral part of the mainchain contract, but it is executed on a sidechain. When smart contract methods are moved from the mainchain to a sidechain, of course, they have to be invoked from the mainchain. When there is a call on a mainchain to a smart contract that is deployed on a sidechain, the mainchain method must redirect the call to the sidechain. However, difficulties arise when state variables are accessed during a method executed on a sidechain—a method deployed on a sidechain does not have access to any state variables stored on the mainchain. For that reason, they need to be fetched from the mainchain and stored on a sidechain, which acts as a cache. Consequently, on the first call to a method deployed on a sidechain, in addition to the call parameters, also included are the blockchain data that will be accessed by the method. At the design time, static analysis of the sidechain code is used to determine which data is provided on the first call to a method so that it would populate the cache on a sidechain. Of course, there will still be cache misses due to the sidechain method referencing objects dynamically.

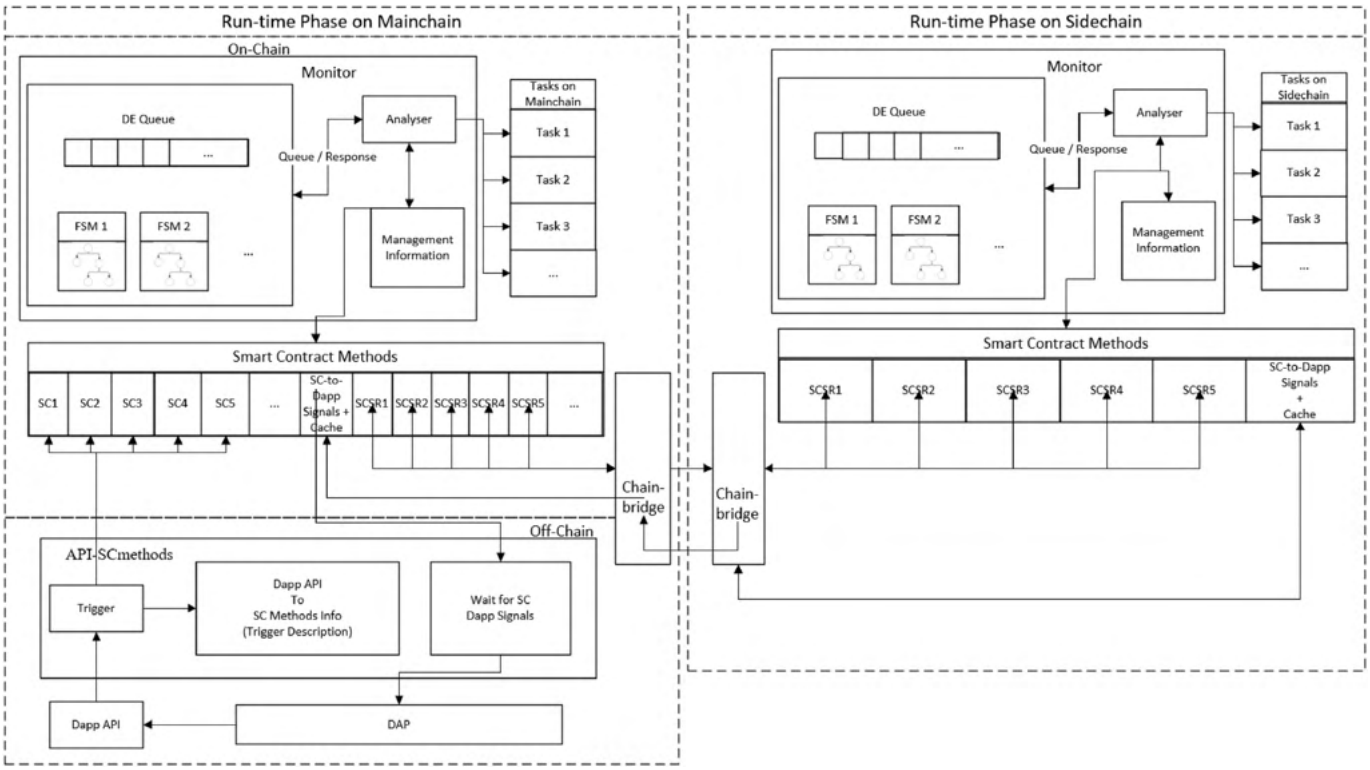


Fig. 10. System architecture with mainchain and sidechain processing.

5.3. Sidechain processing: what, when, how

To process a part of the smart contract off-chain, the following decisions need to be made:

- What should be processed off-chain, that is, which parts, referred to as patterns, of smart contracts should be processed off-chain?
- When should the processing be performed off-chain, that is, under which conditions should sidechain processing be used?
- How to achieve sidechain processing?

5.3.1. Identifying patterns suitable for sidechain processing

The concept of independent subgraphs, which was used to determine the DE-HSM model given a BPMN model, is useful in determining which patterns are suitable for processing off-chain. Processing patterns off-chain causes overhead, some of which is due to the sidechain not having access to the mainchain. Consequently, if a pattern interacts with the main chain methods, the overhead cost of communication between the mainchain and sidechain arises. For that reason, a pattern moved to the sidechain should be independent from the other methods of smart contract executing on the main chain. And independent LSI subgraphs have the property in that once the flow of control enters the LSI subgraph, the flow of control does not exit that subgraph until the computation in that subgraph is completed when the flow of control exits the subgraph. Thus, if the LSI-subgraph processing is deployed on a sidechain, with the exception of the main chain contract passing the application's input as parameters to the sidechain, there is no processing on the mainchain until the execution exits from the independent subgraph executed on a sidechain. As the design phase uses the concept of LSI subgraphs to find the DE-HSM model, each sub-model is represented by an LSI subgraph, and hence, deploying parts of the smart contract to a sidechain is equivalent to deploying DE-HSM sub-models on the sidechain. The developer is thus provided with the choice of which DE-HSM sub-models should be selected for processing on a sidechain.

5.3.2. When to process on a sidechain (deciding to process a pattern off-chain)

Sidechain processing may be cheaper and may be used to decrease the cost of processing a pattern. However, processing a pattern on a sidechain causes overhead due to interaction between the mainchain and the sidechain. Thus, from the cost perspective only, sidechain processing should be used only if its cost, including the overhead cost, is less than processing the pattern on the mainchain. We determine the cost of processing on a sidechain that also includes the overhead cost by actually doing a live test of running the DApp when (a) the smart contract is deployed on the mainchain only and (b) the pattern is deployed on the sidechain and thus measuring all delays, including delays due to overhead. If processing on a mainchain or a sidechain involves an EVM, the cost is also estimated in *Eth* for each of the smart contract methods and is provided to the modeler.

However, another reason to process a pattern on a sidechain may be privacy. A smart contract deployed on the mainchain is such that all participants can explore all data stored on the blockchain by the smart contract; hence, there is no privacy that can be provided so that some collaborations are private, that is, confidential to other actors.[1] For instance, a subset of actors performs a local collaboration represented by a pattern, and that collaboration should be private in that its details (its blockchain data) should not be visible to the other participants in the smart contract who do not participate in that collaboration. In such a case, we should process the pattern on a sidechain, which will provide privacy, and the other actors will not be privy to the pattern's processing details. Of course, all smart contract participants will be able to see any data returned by the sidechain computation that is stored on the mainchain.

[1] It should be noted that our statements made about privacy were of general nature and that there are blockchains that do provide for privacy. For instance, the Hyperledger Fabric enables defining certain data to be private and not visible to all participants.

5.3.3. How to process off-chain

Once it is determined which pattern is to be processed off-chain, the next question is how to facilitate it. A pattern is a set of smart contract methods such that they are invoked either due to a signal from the DApp API or because they collaborate and invoke each other but "within" the pattern. A smart contract method executing on a sidechain is likely to need to know the state of execution and other information stored on the blockchain; however, a method on the sidechain does not have access to the mainchain and its data. Consequently, when transitioning to off-chain execution of a smart contract method, in addition to the parameters passed to the method, the system also needs to find, retrieve, and deliver to the off-chain processing of any mainchain data that the methods on a sidechain read. Similarly, upon execution returning back to the mainchain, any writes to the blockchain that was performed by the off-chain execution of the pattern need to be collected and handed off, together with transition outputs, for recording on the blockchain. Finally, upon completion of a method executed off-chain, before the blockchain is updated with the new state and data values written by the off-chain method, the results of off-chain execution must be reviewed and approved/attested by each of the participants affected by the off-chain computation. As a consequence of the above, we provide interfaces for the following three phases when methods of a pattern are deployed on a sidechain:

Pattern start: Upon the first invocation of an off-chain pattern method, our software tool prepares appropriate data structures to support on-chain/off-chain interaction. The most important is a cache for mainchain data accessed by the pattern executed off-chain. Off-chain code uses a local cache for reading and writing blockchain data; its data structure needs to be prepared. On a cache miss, data are retrieved from the main blockchain and then stored in the off-chain cache using a getter method prepared just for that purpose to retrieve the data from the mainchain. For the cache, we use IPFS as a distributed system for storing and accessing temporary files and data. When blockchain needs to access or examine the data in the cache, TABS allocates a data resource stored on the IPFS by tracking the content address of the corresponding data stored on the IPFS [39].

As mentioned before, a simple static analysis of the pattern methods is used to determine which state variables, that is, blockchain data, should be passed to the sidechain with the first invocation of the pattern so that the cache could be populated with them. Of course, any dynamically accessed data may result in a cache miss.

Pattern middle: After the first invocation, the interaction between the off-chain and on-chain communications is concerned with the provision and return of appropriate parameters and data for methods executed off-chain. Also, on a cache miss to the off-chain cache, data need to be retrieved from the main chain using a getter method to service the cache fault.

Pattern end: Upon completion of the last method that is executed off-chain, which occurs upon a transition from the off-chain pattern execution to the on-mainchain execution, in addition to the returned parameters, the results produced by the off-chain computation that were saved in a cache are collected and are provided to the attestation procedure executed on the mainchain. Furthermore, once the results are approved by attestation, they are recorded on the mainchain. Our default attestation is to deliver the results of sidechain processing to each of the participants collaborating on the sidechain processing, including the hash of the results, and receive from each actor the digitally signed hash code of the result (signed by the private key), thus signifying that the actor accepts the results. If the results are not accepted by an actor, an exception is raised. Of course, the default attestation method may be replaced by an attestation method provided by the developer.

5.4. Sidechain processing control

As sidechain processing may be used for either cost reduction or privacy, our tool, described in the next section, delegates the responsibility for determining which patterns to process on a sidechain to the modeler/developer. The tool displays the list of LSI patterns of the smart contract, DE-HSM sub-models, to the developer, who determines which pattern is or is not processed on a sidechain. For informed decision-making regarding the cost of processing on mainchain *vs.* sidechain, we deploy the smart contract with and without the pattern executing on a sidechain to determine their costs/delays and present them to the developer to assist in the decision-making. An example will be provided in the next section that describes the TABS tool.

6. Tabs tool and evaluation

We developed a tool, called **T**ransforming **A**utomatically **B**PMN Models into **S**mart Contracts (TABS), as a PoC that our approach is feasible and to discover any issues that may arise when a designed system undergoes implementation. We first describe the tool and then its evaluation.

6.1. TABS tool for *t*ransforming *a*utomatically *B*PMN models into *s*mart contracts

Transformation starts with a BPMN model specification expressed in XML as per BPMN specifications [37]. The TABS tool invokes the Camunda software platform [36], whichis used by the developer to create a BPMN model and store it in XML format. Fig. 11 shows a partial screen of the tool when creating a BPMN model for the supply-chain application. Once the modeler creates a BPMN model and stores it in an XML format, it is used as input to transformations using tabs appearing under the title. The first two tabs deal with BPMN modeling, one for a BPMN choreography, and one for a model expressed in core BPMN elements. Note that the term choreography here refers to the BPMN concept of choreography that shows "conversations" that eventually need to be elaborated upon by the modeler into a model expressed using BPMN's core elements. Thus, the term "choreography", as used in BPMN, is not to be confused with the term choreography as used in previous work on transforming BPMN models into smart contracts, in which the term refers to the choreography of processes into which a BPMN model is transformed and wherein the choreography is a major part of the job mediator's job of the system architecture of Fig. 2.

Once the BPMN model is developed, the modeler's interaction with the tool consists of (i) guiding the transformation of the BPMN model into a smart contract using tabs and (ii) supplying the template methods for BPMN tasks with code. We facilitate the creation of a smart contract and sidechain smart contract(s) either for Ethereum or Hyperledger Fabric blockchains. However, the tool does not facilitate the automated creation of the underlying blockchain Fabric itself. In fact, the tool supports using two blockchain types: (i) Hyperledger Fabric and (ii) Ethereum or EVM-based blockchain such as Quorum. Either type can be used for the mainchain or for a sidechain(s), wherein the mainchain smart contract invokes methods of the smart contract deployed on a sidechain.

Fig. 12(a) shows a screenshot after the supply-chain BPMN model, as shown in Fig. 3, was transformed into the DE-HSM model having the four LSI subgraphs. It shows the BPMN graph and its LSI subgraphs that have been identified by the TABS tool. Fig. 12(b) was not generated by the tool but was included to show the BPMN representation of the corresponding LSI subgraphs of Fig. 12(a).

The modeler/developer is also involved in the decision-making on which of the independent subgraph patterns should be deployed on a sidechain in the form of a separate smart contract that interacts with the main contract deployed and executed on the main blockchain. The modeler then selects which of the Ethereum or Hyperledger Fabric is to be used for each of the mainchain and a sidechain—for testing purposes, we have local blockchains, one for each of the Ethereum and Hyperledger Fabric. On Hyperledger Fabric, we also have channels prepared that are used for the deployment of smart contracts for both the mainchain and sidechains. For sidechains compatible with Ethereum, we use the

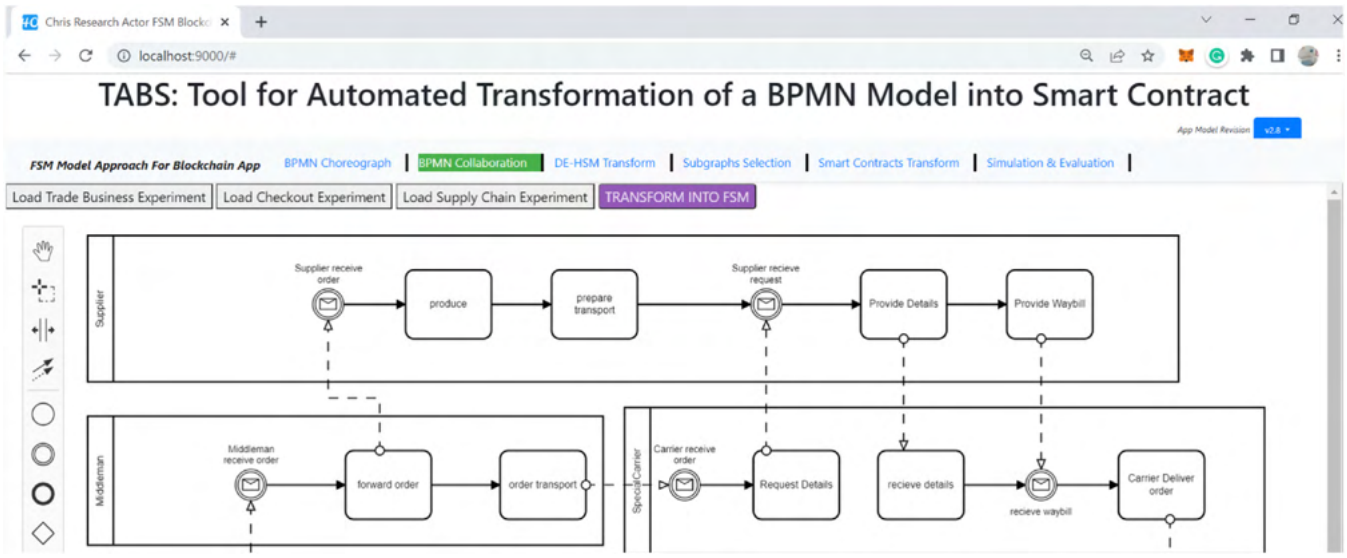


Fig. 11. TABS tool invoking the Camunda platform to create a BPMN model of the supply chain application.

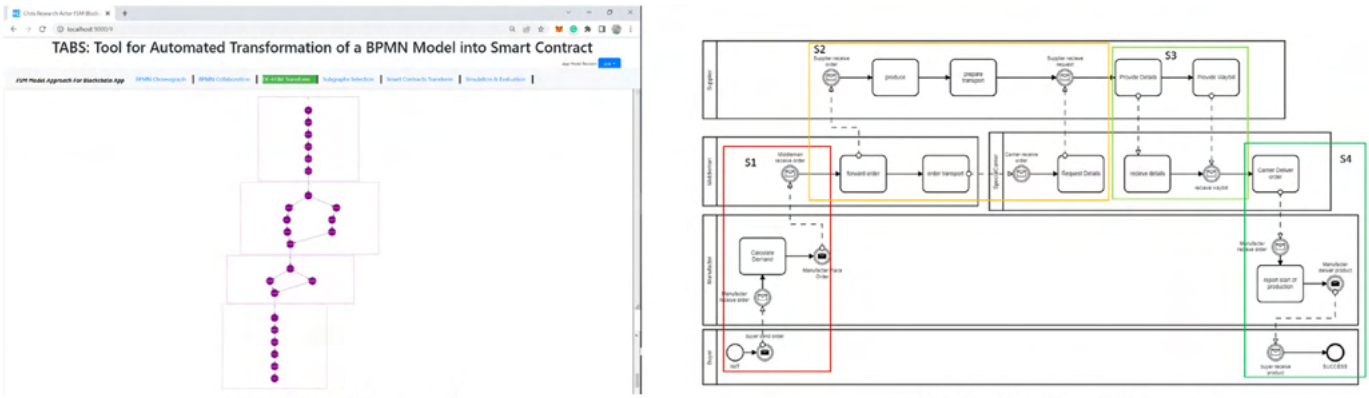


Fig. 12. LSI subgraph after transformation into a DE-HSM model.

Quorum sidechain.

Once the selection is made by the developer, the model is transformed into the methods of smart contract(s) that are deployed and executed. The modeler can examine the generated system by stepping through execution message-by-message, while the tool shows the progress graphically by showing the change to states of the individual FSMs representing the DE-FSM sub-models. This feature is helpful in testing and manual verification. Delays are also shown as execution proceeds step by step.

To evaluate the latency and cost of the overall execution, the tool facilitates the execution of each method while measuring its latency and cost. It should be noted that we show the USD cost of execution in Wei (or GWei) [45] for Ethereum or EVM-based blockchains/sidechains only. Latency and costs are calculated for two cases: (i) when all methods of smart contracts are deployed on the mainchain only, and (ii) when execution proceeds on the mainchain with the selected patterns deployed on a sidechain. Thus, the developer can compare the latency and costs when execution is on the mainchain only and when the selected patterns are processed on a sidechain. Fig. 13 exhibits a screenshot showing the cost of execution on a local Ethereum mainchain and a local Quorum sidechain, where the cost of execution is in GWei and is derived by adding the cost of individual Solidity instructions of the smart contract methods using Web3.js [46]. As far as the cost is concerned, for Ethereum, we estimate the cost in GWei units using Web3.js, which is certainly useful to the modelers. The cost for the case when Hyperledger Fabric is used depends directly on the cost of the underlying blockchain infrastructure, and we have not tackled yet how to derive it. As in the case of the public Ethereum blockchain, blockchain as a service for Hyperledger Fabric is not cheap, and it depends on the blockchain configuration in terms of the network nodes, network bandwidth, etc.

6.2. *Evaluation*

As mentioned above, we developed the TABS tool as a PoC so that our approach is feasible and to explore its properties, such as latency and cost. We first outline our evaluation method and then discuss the use cases we tested, transformation correctness, latency, and cost.

6.2.1. *Evaluation method*

The TABS tool is a typical two-tier application. The frontend was mainly written in NodeJS, which focuses on tasks such as:

a. Establish a distributed message channel with backend IPFS nodes.
b. Provide the canvas to the user for composing the BPMN diagram; this was implemented by BPMN.io [47].
c. Transform the BPMN diagram to DE-HSM mode. This part was powered by the GraphViz JS library [48].

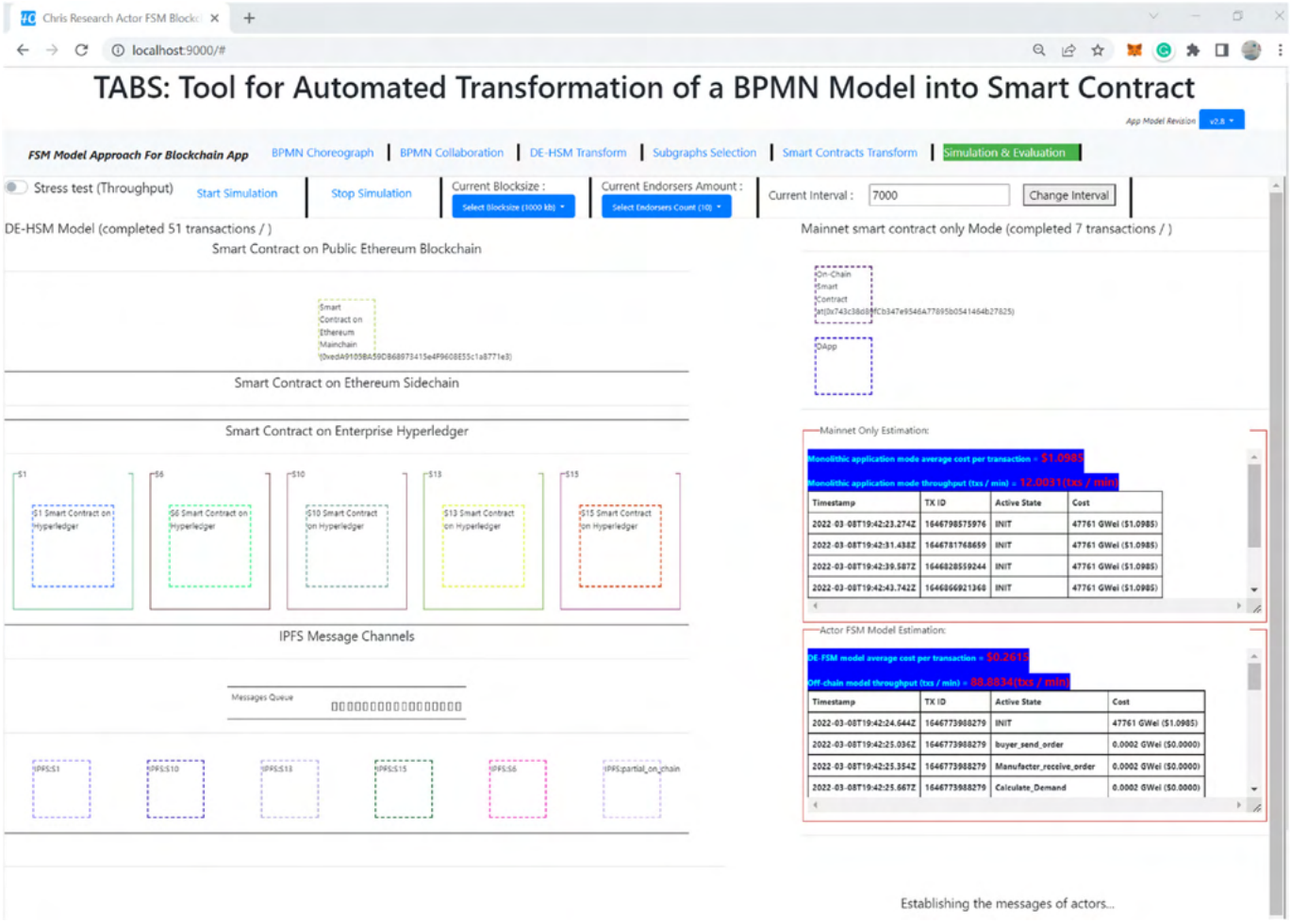


Fig. 13. Stress testing smart contract method execution of bulk business processes.

d. Provide the Ethereum Solidity abstract interfaces and Hyperledger NodeJS abstract interfaces for smart contract transformation. The interfaces were coded in Solidity and NodeJS, respectively [49,50].
e. Compile the smart contract with Ethereum powered by NodeJS worker threads [51].
f. Compile the smart contract with IBM blockchain platform developer tools [52].
g. Deploy the Ethereum smart contract with EtherJS and Web3.js [46, 53].
h. Deploy the Hyperledger Chaincode with the Microfab API [54].
i. Establish the IPFS message channel with the IPFS API [55].
j. Implement evaluation and simulation process with NodeJS [56].

We use three cloud servers provided by DigitalOcean cloud [57]. Each server is equipped with 2 CPUs, 4 GB memory, and 80 GB disk space. The operating system running on the servers is Ubuntu. We installed, configured, and ran Ganache-CLSI on each server with different parameters to emulate a realistic working environment. The mainchain blockchain was configured with different parameters for block time, endorsers, and block size, but with values that are close to the public Ethereum blockchain, while the sidechain networks were configured as Quorums for Ethereum smart contract and Microfab for Hyperledger Chaincode. Moreover, our TABS tool allows testers to deploy smart contracts on actual public Ethereum and Ropsten sidechains as long as users have sufficient cryptocurrency on those blockchains [58]. Furthermore, each server is an IPFS node located in a preconfigured IPFS private cluster so that each participant has dedicated access to the corresponding IPFS node and data space.

The evaluation process of a use case starts with BPMN modeling. At the end of modeling, TABS automatically transforms the BPMN model to a DE-HSM model during the design phase. The modeler then provides the code for the tasks of the BPMN elements. The DE-HSM model is presented to the modeler together with information on the cost of execution on the mainchain only and on the mainchain with sidechain processing of the selected subgraphs. The modeler uses this information to select which of the LSI subgraphs are to be deployed and processed on a sidechain, and then chooses which of the Ethereum Solidity and Hyperledger chaincodes are to be used for the smart contract generation on the mainchain and a sidechain. The generated code for the smart contract(s) may be reviewed by the developer.

Once the compiled smart contracts are deployed on blockchain networks, the modeler can choose which evaluation process should be used for the run-time phase. The modeler can observe the invocation of the smart contract methods. The TABS tool allows the modeler to evaluate the deployed model by an input stream that simulates the expected input from the DApp as it would be submitted by the DApp to the DApp interface. Measured are delays per smart contract call, for completion of executing a sub-model, or per execution of the DApp application (on the BPMN diagram, from the Start event to the End event).

6.2.2. *Use cases*

For our evaluation purpose, we used three different use cases.

1. Supply Chain: This use case was adopted from Ref. [16] and was discussed throughout this paper as a running example (see Figs. 3 and 7). The model has ten tasks and four gateways. The supply chain use

case begins with the buyer issuing a new order. Once the manufacturer receives the order, she/he calculates demand and places an order with a middleman. The middleman concurrently sends the order to the supplier, and orders are transported from the carrier. The producer fabricates the product and prepares it for transport. The carrier, upon receiving the request from the middleman, requests details from the supplier. The supplier provides the details to the carrier and then prepares and provides the waybill to the carrier. Upon receiving details about the product and the waybill, both from the supplier, the carrier delivers the order to the manufacturer. Upon receiving the order, the manufacturer starts the production, and when that is finished, it delivers the product to the buyer, who receives the order.

2. Order Process: This process is adopted from Ref. [59], and its BPMN diagram is shown in Fig. 14(a). It is a relatively straightforward use case with five tasks and two gateways. The use case starts with the customer preparing an order that is delivered to order handling. Order handling checks the order and then concurrently (i) confirms the order with the customer and (ii) initiates shipment preparation by the shipper. The shipper prepares the order for shipment and then sends it to the customer. Upon receiving the order confirmation and the order from the shipper, the customer accepts the order, which completes the workflow. Fig. 14(b) shows the BPMN diagram with the three found LSI subgraphs identified by the colored rectangles.
3. Trade: We reported on this use case in some detail in our previous research [43]. The use case was adopted from Ref. [60], and it contains 19 tasks and five gateways. As the BPMN diagram of the application would be too busy, we show in Fig. 15 its state diagram representation with some of the independent subgraphs also shown. The model shows that the product is posted for sale, and after the offer is made, the buyer and seller negotiate the price. Once an agreement is reached on the price, a contract is prepared and signed that stipulates escrow deposit and the matter of delivery. Once the buyer makes a deposit to an escrow account, the shipment proceeds to the port. The shipment includes crossing borders, and hence, involves customs. Afterwards, the shipment is loaded on a ship, delivered to a port, and then processed at the destination customs. It is then unloaded from ship to port, picked up by the buyer, payment terms are executed, and finally, the escrow deposit is returned.

6.2.3. Latency and cost

We now report on the evaluation of the three use cases in terms of latency and cost. However, the cost is derived only for Ethereum using Web3.js [46], which estimates the cost of each instruction in a smart contract in Wei. In this section, we simply report observed values as we are unable to do a comparison with the other reported work because hardware and software configurations and environments vary greatly. We report observed values for each case when Ethereum is used and when the Hyperledger Fabric is used. We comment on our observations when sidechains are used for off-chain processing in the next subsection.

a) Latency

The TABS tool can measure various latencies, ranging from one FSM state transition, through latency for executing a sub-model or a smart contract method, to latency for the whole DApp. Fig. 16 shows the delays in milliseconds for each use case execution and for each Ethereum and Hyperledger Fabric. Due to the more complex consensus algorithm, Ethereum exhibits much higher delays than Hyperledger.

b) Cost

Costs per method and the overall cost can also be estimated automatically through TABS by using web3.js [46], which enables the calculation of the cost for each instruction. The cost in GWei for each of the three use cases is shown in Table 1 for each of the three use cases. The table also shows the cost, in USD at rates prevailing at the time of measurements, of one application workflow for each use case for the public Ethereum. As the Hyperledger Fabric is a private blockchain, the cost is primarily due to hardware and network costs.

6.2.4. Correctness (non-conforming vs. conforming traces)

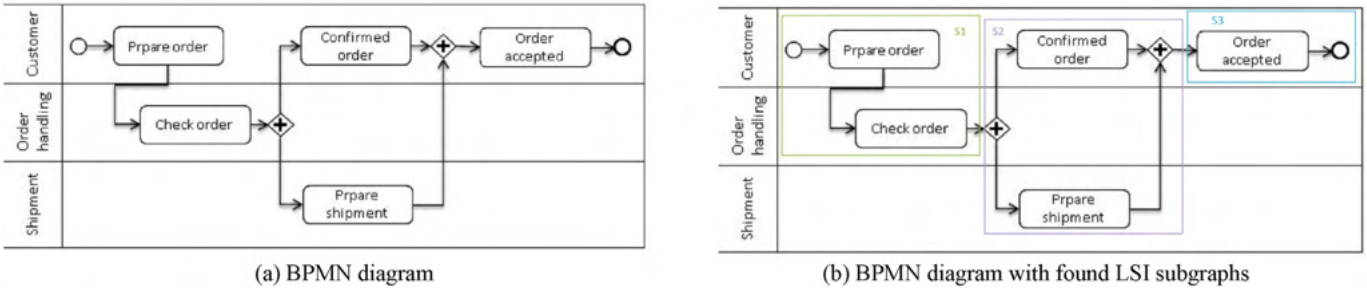
In published research on transforming BPMN models to smart contracts, reporting was done on the correctness of the smart contract by recognizing conforming and non-conforming traces, wherein a trace is an input stream from the DApp. A conforming trace represents input from the DApp that is correct in terms of the correctness of the input provided by the DApp at appropriate points of a smart contract execution. The smart contract should recognize incorrect input provided by the DApp. We tested the smart contract implementation and reported that 100% accuracy was achieved in recognizing conformant and non-conformant input sequences/traces. This is not surprising—after all, the smart contract is built through DE-HSM modeling that transforms the BPMN model into a DE-HSM model that, in essence, is formed by a 1-to-1 mapping between the BPMN model and the DE-HSM model as a collection of DE-FSM sub-models, wherein each sub-model corresponds directly to the BPMN diagram when viewed as a DAG. Thus, in the absence of errors in the TABS tool, the recognition of correct and incorrect sequencing in input should be 100%.

6.2.5. Sidechain processing

One of the distinguishing features of our approach is that we can move blockchain patterns, represented as individual DE-HSM sub-models, to the sidechain for either of the following two purposes or their combination: (i) reducing the cost of processing and (ii) privacy.

a) *Sidechains for cost reduction*

Deploying patterns for processing on a sidechain for the purposes of reducing the cost of processing is useful only when sidechain processing is cheaper in comparison to mainchain processing. This is important for



(a) BPMN diagram (b) BPMN diagram with found LSI subgraphs

Fig. 14. BPMN diagram for the Order Proces use case (adopted from Ref. [60]).

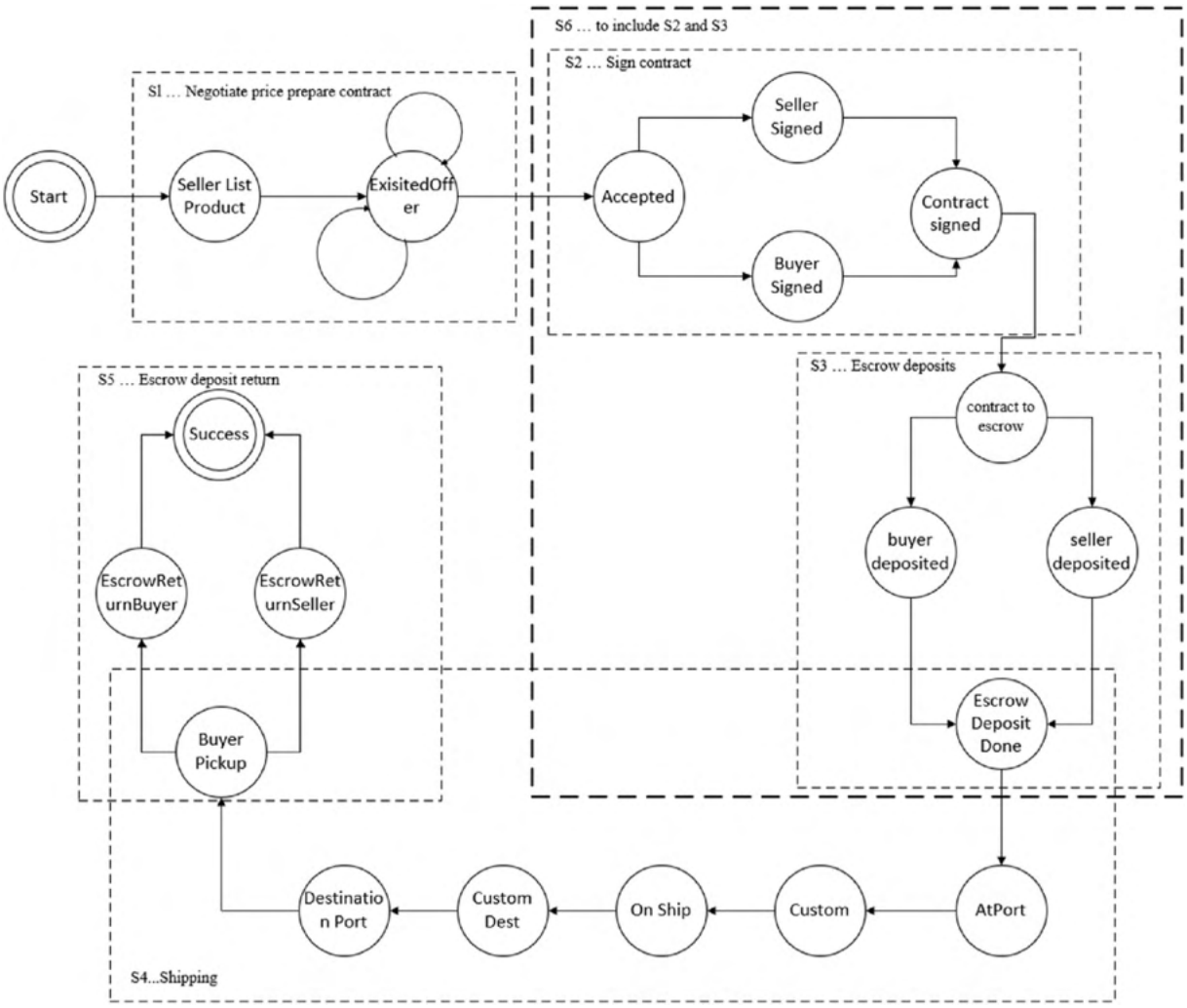


Fig. 15. State transition diagram for the trade business use case (adopted from Ref. [44]).

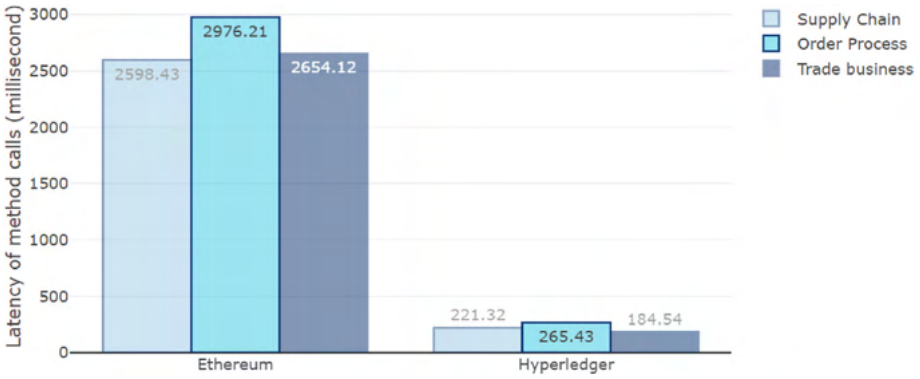


Fig. 16. Latency for each of the three use cases for Ethereum and for Hyperledger Fabric.

Table 1
Cost of processing for Ethereum environment.

	Gas (GWei)	Cost (USD)
Supply Chain	95,854	0.2615
Order Process	57,512	0.1569
Trade Business	182,195	0.4945

any smart contract targeted for deployment on a blockchain—blockchains are expensive relative to centralized systems. As an example, processing on the public Ethereum blockchain is much more expensive than on the Quorum chain, and hence, sidechain processing is attractive for reducing the cost. For Hyperledger Fabric, using channels as sidechains for cost reduction does not make sense, as a mainchain on a Hyperledger is just one of the many channels. Hyperledger Fabric may be used as the mainchain while some other blockchain is used for a

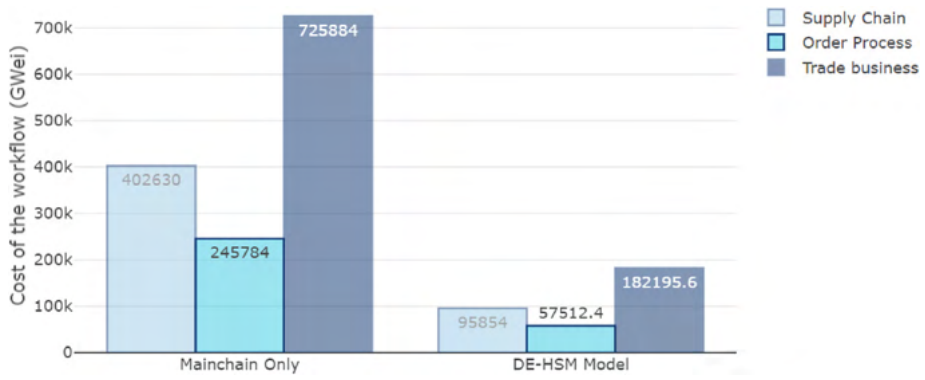


Fig. 17. Ethereum cost of processing on mainchain only *vs.* processing on mainchain with sub-models on a sidechain.

sidechain, as long as it (sidechain blockchain) is cheaper than the mainchain.

For each use case, we calculated the cost of processing one execution of a smart contract. For each use case, Fig. 17 shows the cost of processing on the Ethereum mainchain only and then when a Quorum sidechain is used with the mainchain. When sidechain processing is used, for each use case, we select all sub-models to be processed on a sidechain. The selected sub-models are identified by either colored or dashed rectangles in Figs. 7, 12 and 15. Fig. 19 shows that because the Quorum cost of processing is much lower than the cost of professing on the mainchain, processing off-chain is very advantageous.

We also compare the latency of processing, which is shown in Fig. 18, for the mainchain processing on Ethereum and for the mainchain with sidechain processing on Quorum for each of the use cases. Latency is also reduced by sidechain processing. Similarly, we used sidechain processing for Hyperledger Fabric by deploying the main smart contract on one Hyperledger channel and a sidechain on another channel of the same Hyperledger Fabric blockchain. Not surprisingly, latency with sidechain processing is higher than just processing on the mainchain only, as shown in Fig. 19. Sidechain processing on a Hyperledger is achieved by using another Hyperledger channel that incurs the same cost as processing on the mainchain, which is also a Hyperledger channel. Due to the overhead to facilitate sidechain processing, latency is higher when sidechain is used.

b) *Sidechains for privacy*

As discussed already, sidechains may also be used for privacy. In more complex smart contracts, such as the trade use case, certain parts of the contract should not be visible to all participants of the smart contract. For instance, after the buyer and seller agree on the price, the price should not be visible to some of the other actors. By choosing to deploy and execute an independent subgraph pattern, i.e., a sub-model, on a sidechain, privacy is also provided, as only the actors who participate in the sidechain sub-model execution can observe all data pertaining to that sidechain processing, while the other actors do not have access to that data. However, all actors have access to the results of computation that are stored on the mainchain. As was already noted before, the above statements are subject to privacy features of individual blockchains, as some blockchains, such as Hyperledger Fabric, do support private data on a channel that is not visible to other participants.

7. Concluding remarks

This paper contributes to the research on generating smart contracts from a BPMN model. As in previously reported research, we minimize the modeler's interaction with the system in that once the BPMN model is input, the modeler simply guides the tool in the generation of the smart contract methods. The modeler guides the tool in the transformation process and provides the necessary information to the tool, such as the code for template methods implementing BPMN task elements or information on which parts of the smart contract should be deployed on a sidechain. In addition, the modeler/developer can use the tool to explore the execution properties of the methods of the smart contract. Currently, our tool provides the creation and deployment of contracts for either Ethereum or Hyperledger blockchains.

The following two features are the major differences from the work reported by others.

- We analyze the BPMN diagram and use DE-HSM modeling to identify patterns, referred to as LSI subgraphs, that are localized in that once execution of the pattern starts, it remains local to the pattern until there is an exit from that pattern. Furthermore, each pattern has only one “entry” and one “exit” point. Such patterns naturally lend themselves to decomposition purposes that our approach exploits. We use such patterns for forming the sub-models, wherein the functionality of each sub-model is represented by an FSM or concurrent FSMs. The choreography (not a BPMN term) of processes, that is, the workflow determining which sub-models are executed in which sequence, is guided by the interconnections amongst the sub-models, while the functionality of each sub-model is represented by an FSM, or possibly a number of concurrent FSMs. The interconnection of the sub-models is represented by the DE-HSM model. Consequently, as the DE-HSM is a model that is an equivalent representation of a BPMN model, the correctness of representing the BPMN model using the DE-HSM model is assured.
- We provide for the transformation of sub-models, selected by the modeler, into a separate contract that is deployed and executed on a sidechain, while the main smart contract interacts with the DApp and invokes these sidechain methods as is appropriate. Sidechain processing may be chosen by a modeler (i) to reduce the cost of processing if sidechain processing is cheaper than processing on a mainchain (e.g., processing on a Quorum sidechain in conjunction with the main contract being deployed on the public Ethereum blockchain) or (ii) to support privacy by processing a pattern on a sidechain.

We made several simplifying assumptions on the BPMN model, such as limited looping/repeating or parallel sub-processes, and no feedback loops in the BPMN model that cannot be expressed as looping or concurrent sub-processes that need to be addressed. The tool's limitation is that, prior to the execution of a looping sub-process, the number of iterations needs to be known. The assumption on the parallel sub-processes is more restrictive in that the maximum number of parallel streams to execute a parallel sub-process needs to be known at the time of transforming the BPMN model into the methods of a smart contract. We are investigating approaches to reduce or remove these limitations.

The BPMN standard defines a transaction on a sub-process, in which

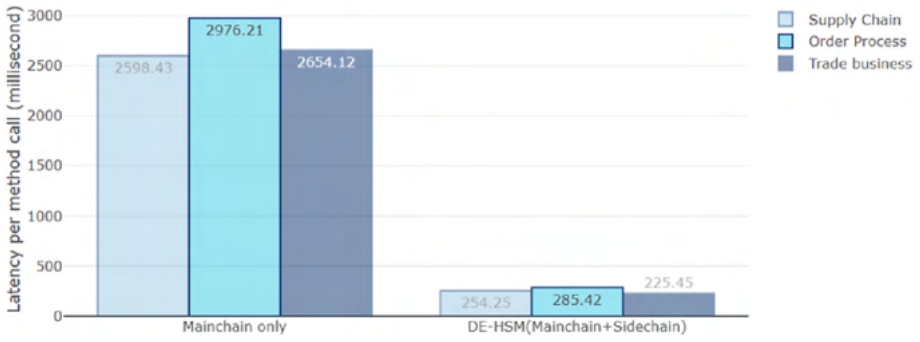


Fig. 18. Ethereum latency of processing on mainchain *vs.*

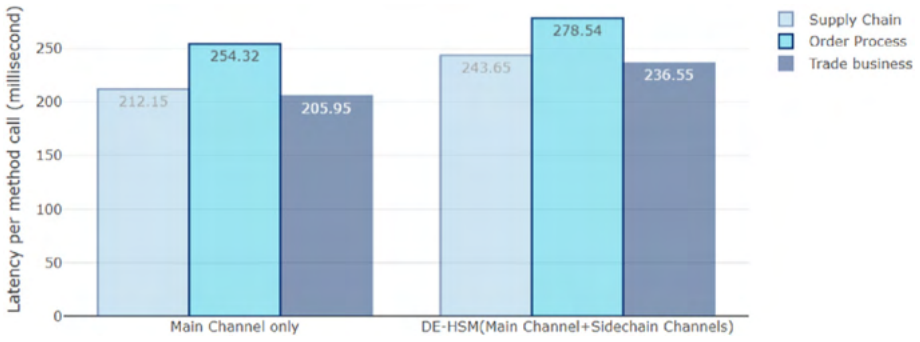


Fig. 19. Hyperledger latency of processing of all on one channel *vs.*sub-models on the Quorum sidechain. processing sub-models on another channel.

case the whole sub-process must be completed or any activity of the sub-process must be undone by a compensating transaction if the sub-process is unable to be completed successfully. As was already discussed before, a blockchain also has a transaction that is defined as any set of ledger updates made by the execution of *a single smart contract* method. However, a trade transaction is long-term and may include many activities, activities performed by a subset of the application participants on a subset of data used by the application. And these three different concepts do not align. How to support long-term trade transactions, which span a number of calls to the smart contract methods, is still an open question that we are currently addressing. There are also some of the BPMN elements, such as error, escalation, and multiple parallel attributes, that our TABS system does not support, but we are working on their incorporation into the system.

Currently, we provide a demo and limited-term online access to the tool upon request that can be made by emailing Chris.Liu@dal.ca. In the near future, we plan to prepare appropriate documentation to facilitate future potential collaborations.

Of course, our current implementation is only a PoC, and many issues still need to be addressed. For instance, as contract execution consists of the removal of events from the DE queue and processing them, issues of efficiency and scalability need to be addressed. The use of a global logical clock to timestamp events, which are stored in the timestamp-priority ordered queue DE queue, may also lead to scalability issues that need to be investigated. We note that the properties of trade applications are such that the speed of processing is not a priority as the transactions in this vertical are relatively long-term. The issue of scalability may arise due to the increase in the complexity of applications and in the number of applications that need to be supported.

Author contributions

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

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] J. Eberhardt, S. Tai, On or off the blockchain? Insights on off-chaining computation and data, in: F. De Paoli, S. Schulte, E. Broch Johnsen (Eds.), Service-Oriented and Cloud Computing, Springer, Cham, 2017, pp. 3–15, https://doi.org/10.1007/978-3-319-67262-5_1.

[2] J. Poon, V. Buterin, Plasma: Scalable Autonomous Smart Contracts, 2017. https://plasma.io/plasma.pdf. (Accessed 12 November 2020).

[3] D. Yang, C. Long, H. Xu, et al., A review on scalability of blockchain, in: Proceedings of the 2020 the 2nd International Conference on Blockchain Technology (ICBCT'20), ACM, 2020, pp. 1–6, https://doi.org/10.1145/3390566.3391665.

[4] Shard Chains | Ethereum.Org. ethereum.Org, 2021. https://ethereum.org/en/eth2/shard-chains/. (Accessed 26 March 2022).

[5] 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.

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

[7] 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. Software 174 (2021), 110891, https://doi.org/10.1016/j.jss.2020.110891.

[8] R. Belchior, A. Vasconcelos, S. Guerreiro, et al., A survey on blockchain interoperability: past, present, and future trends, ACM Comput. Surv. 54 (8) (2021) 168, https://doi.org/10.1145/3471140.

[9] 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.

[10] 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.

[11] C. Lauster, P. Klinger, N. Schwab, et al., Literature review linking blockchain and business process management, Proc. 15th Int. Conf. Wirtschaftsinformatik..GITO. (2020), https://doi.org/10.30844/wi_2020_r10-klinger.

[12] 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 (PoEM'21 Forum), CEUR., 2021, pp. 11–20.
[13] P. Tolmach, Y. Li, S.-W. Lin, et al., A survey of smart contract formal specification and verification, ACM Comput. Surv. 54 (7) (2021) 148, https://doi.org/10.1145/3464421.
[14] S. Bragagnolo, H. Rocha, M. Denker, et al., SmartInspect: solidity smart contract inspector, in: Proceedings of the 2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE), IEEE, 2018, pp. 9–18, https://doi.org/10.1109/IWBOSE.2018.8327566.
[15] J. Mendling, I. Weber, W.M.P. van der Aalst, et al., Blockchains for business process management - challenges and opportunities, ACM Trans. Manag. Inf. Syst. 9 (1) (2018) 4, https://doi.org/10.1145/3183367.
[16] I. Weber, X. Xu, R. Riveret, et al., Untrusted business process monitoring and execution using blockchain, in: M. La Rosa, P. Loos, O. Pastor (Eds.), Business Process Management, Springer, Cham, 2016, pp. 329–347, https://doi.org/10.1007/978-3-319-45348-4_19.
[17] O. López-Pintado, L. García-Bañuelos, M. Dumas, et al., Caterpillar: a business process execution engine on the Ethereum blockchain, Software Pract. Ex. 49 (7) (2019) 1162–1193, https://doi.org/10.1002/spe.2702.
[18] A.B. Tran, Q. Lu, I. Weber Lorikeet, A model-driven engineering tool for blockchain-based business process execution and asset management, in: Proceedings of the Dissertation Award and Demonstration, Industrial Track at BPM 2018, CEUR, 2018, pp. 56–60.
[19] O. López-Pintado, M. Dumas, L. García-Bañuelos, et al., Dynamic role binding in blockchain-based collaborative business processes, in: P. Giorgini, B. Weber (Eds.), Advanced Information Systems Engineering, Springer, Cham, 2019, pp. 399–414, https://doi.org/10.1007/978-3-030-21290-2_25.
[20] 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.
[21] C. Di Ciccio, A. Cecconi, M. Dumas, et al., Blockchain support for collaborative business processes, Informatik-Spektrum 42 (2019) 182–190, https://doi.org/10.1007/s00287-019-01178-x.
[22] F. Loukil, K. Boukadi, M. Abed, et al., Decentralized collaborative business process execution using blockchain, World Wide Web 24 (2021) 1645–1663, https://doi.org/10.1007/s11280-021-00901-7.
[23] Q. Lu, A.B. Tran, I. Weber, et al., Integrated model-driven engineering of blockchain applications for business processes and asset management, Software Pract. Ex. 51 (5) (2021) 1059–1079, https://doi.org/10.1002/spe.2931.
[24] L. Spalazzi, F. Spegni, A. Corneli, et al., Blockchain Based Choreographies: the Construction Industry Case Study. Concurrency and Computation: Practice and Experience, 2021, e6740, https://doi.org/10.1002/cpe.6740.
[25] 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.
[26] 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.
[27] D. Harel Statecharts, A visual formalism for complex systems, Sci. Comput. Program. 8 (3) (1987) 231–274, https://doi.org/10.1016/0167-6423(87)90035-9.
[28] A. Girault, B. Lee, E.A. Lee, Hierarchical finite state machines with multiple concurrency models, IEEE Trans. Comput. Aided Des. Integrated Circ. Syst. 18 (6) (1999) 742–760, https://doi.org/10.1109/43.766725.
[29] M. Yannakakis, Hierarchical state machines, in: J. van Leeuwen, O. Watanabe, M. Hagiya, et al. (Eds.), Theoretical Computer Science: Exploring New Frontiers of Theoretical Informatics, Springer, Berlin, Heidelberg, 2000, pp. 315–330, https://doi.org/10.1007/3-540-44929-9_24.
[30] C.A.R. Hoare, Communicating sequential processes, Commun. ACM 21 (8) (1978) 666–677, https://doi.org/10.1145/359576.359585.
[31] C.G. Cassandras, Discrete Event Systems: Modeling and Performance Analysis, Irwin, Homewood IL, 1993.
[32] C. Liu, P. Bodorik, D. Jutla, Blockchain Privacy, Scalability, and Separation of Concerns Using Multi-Modal Modeling, 2021. Access, https://web.cs.dal.ca/~gang/ICBTA/. (Accessed 26 March 2022).
[33] 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.
[34] L. Dikmans, Transforming BPMN into BPEL: why and how. https://www.oracle.com/technical-resources/articles/dikmans-bpm.html, 2008. (Accessed 26 March 2022).
[35] R.M. Dijkman, M. Dumas, C. Ouyang, Semantics and analysis of business process models in BPMN, Inf. Software Technol. 50 (12) (2008) 1281–1294, https://doi.org/10.1016/j.infsof.2008.02.006.
[36] Camunda Modeler, Design business processes and decision models. https://camunda.com/platform/modeler/. (Accessed 29 March 2022).
[37] Flowable Open Source Documentation, BPMN 2.0 introduction. https://www.flowable.com/open-source/docs/bpmn/ch07a-BPMN-Introduction. (Accessed 29 March 2022).
[38] BPMN 2.0 Symbols: a complete guide with examples. https://camunda.com/bpmn/reference/. (Accessed 28 October 2021).
[39] Object–relational impedance mismatch. Wikipedia. https://en.wikipedia.org/wiki/Object%E2%80%93relational_impedance_mismatch. (Accessed 26 March 2022).
[40] M. Steichen, B. Fiz, R. Norvill, et al., Blockchain-based, decentralized access control for IPFS, in: Proceedings of the 2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), IEEE., 2018, pp. 1499–1506, https://doi.org/10.1109/Cybermatics_2018.2018.00253.
[41] What is BPMN?. https://bpmn.gitbook.io/bpmn-guide/what-is-bpmn. (Accessed 26 March 2022).
[42] C.G. Liu, FSM for Modeling for Off-Blockchain Computation, MS Thesis. Dalhousie University, Halifax, Nova Scotia, Canada, 2021.
[43] P. Bodorik, C.G. Liu, D. Jutla, 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 (ICBCT '21), ACM, 2021, pp. 28–34, https://doi.org/10.1145/3460537.3460565.
[44] C. Liu, P. Bodorik, D. Jutla, A tool for moving blockchain computations off-chain, in: Proceedings of 2021 3rd ACM International Symposium on Blockchain and Secure Critical Infrastructure (BSCI'21), ACM, 2021, pp. 103–109, https://doi.org/10.1145/3457337.3457848.
[45] C. Tardi, What is gwei? The cryptocurrency explained. https://www.investopedia.com/terms/g/gwei-ethereum.asp. 2022. (Accessed 29 March 2022).
[46] web3.js - Ethereum JavaScript API. https://web3js.readthedocs.io/. (Accessed 26 March 2022).
[47] bpmn.io · GitHub, 2022. https://github.com/bpmn-io. (Accessed 26 March 2022).
[48] Graphviz. https://graphviz.org/. (Accessed 29 March 2022).
[49] Solidity 0.8.13 documentation. https://docs.soliditylang.org/en/v0.8.13/. (Accessed 29 March 2022).
[50] Hyperledger fabric SDK for Node.js. https://hyperledger.github.io/fabric-sdk-node/. (Accessed 29 March 2022).
[51] Worker threads | Node.js v17.7.1 Documentation. https://nodejs.org/api/worker_threads.html. (Accessed 26 March 2022).
[52] IBM cloud docs. https://cloud.ibm.com/docs/blockchain?topic=blockchaindevelop-vscode. (Accessed 26 March 2022).
[53] ethers. Documentation. https://docs.ethers.io. (Accessed 26 March 2022).
[54] IBM-Blockchain, Microfab is a containerized Hyperledger Fabric runtime for use in development environments. https://github.com/IBM-Blockchain/microfab. (Accessed 26 March 2022).
[55] IPFS cluster consistency model. https://discuss.ipfs.tech/t/ipfs-cluster-consistency-model/6666, 2019. (Accessed 26 March 2022).
[56] Node.js. https://nodejs.org/en/. (Accessed 29 March 2022).
[57] DigitalOcean – the developer cloud. www.digitalocean.com, 2020. (Accessed 26 March 2022).
[58] etherscan.io. TESTNET ropsten (ETH) blockchain explorer. https://ropsten.etherscan.io/. (Accessed 29 March 2022).
[59] A. Fleischmann, W. Schmidt, C. Stary, (Re-)Justifying BPM: a quest for the interaction turn reviewing subject-oriented BPM, in: Proceedings of the 2013 IEEE 15th Conference, IEEE, 2013, https://doi.org/10.1109/CBI.2013.40.
[60] A. Asgaonkar, B. Krishnamachari, Solving the buyer and seller's dilemma: a dual-deposit escrow smart contract for provably cheat-proof delivery and payment for a digital good without a trusted mediator, in: Proceedings of the 2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC), IEEE, 2019, pp. 262–267, https://doi.org/10.1109/BLOC.2019.8751482.