frontiers | Frontiers in Blockchain

TYPE Original Research
PUBLISHED 14 September 2023
DOI 10.3389/fbloc.2023.1248962

OPEN ACCESS

EDITED BY
Kamel Barkaoui,
Conservatoire National des Arts et Métiers (CNAM), France

REVIEWED BY
Marino Miculan,
University of Udine, Italy
Laid Kahloul,
University of Biskra, Algeria

*CORRESPONDENCE
Md Tauseef Alam,
tauseef_2121cs04@iitp.ac.in
Raju Halder,
halder@iitp.ac.in
Abyayananda Maiti,
abyaym@iitp.ac.in

RECEIVED 27 June 2023
ACCEPTED 23 August 2023
PUBLISHED 14 September 2023

CITATION
Alam MT, Halder R and Maiti A (2023), Formal verification of the pub-sub blockchain interoperability protocol using stochastic timed automata. *Front. Blockchain* 6:1248962. doi: 10.3389/fbloc.2023.1248962

Formal verification of the pub-sub blockchain interoperability protocol using stochastic timed automata

Md Tauseef Alam*, Raju Halder* and Abyayananda Maiti*

Department of Computer Science and Engineering, Indian Institute of Technology Patna, Patna, India

In recent times, the research on blockchain interoperability has gained momentum, enabling the entities from different heterogeneous blockchain networks to communicate with each other seamlessly. Amid the proliferation of blockchain ventures, for ensuring the correctness of inter-blockchain communication protocols, manual checking and testing of all the potential pitfalls and possible inter-blockchain interactions are rarely possible. To ameliorate this, in this paper, we propose a systematic approach to model and formally verify the real-time properties of the pub-sub interoperability protocol, with a special focus on message communication through API calls among publishers, subscribers, and brokers. In particular, we use stochastic timed automata for its modeling, and we prove its correctness with respect to a number of relevant properties using model checking—more specifically, the UPPAAL-SMC model checker. To the best of our knowledge, this is the first proposal of its kind to formally verify the blockchain pub-sub interoperability protocol using model checking.

KEYWORDS

blockchain, chaincode, interoperability, stochastic timed automata, model checking, UPPAAL-SMC

1 Introduction

Blockchain technology (Nakamoto, 2008) has dramatically gained momentum within a decade with its evolutionary transformation from blockchain 1.0 to blockchain 4.0 (Khan et al., 2019). We are witnessing its adoption at a large scale in almost every sphere of the cyber technical world, ranging from simple (e.g., gaming and education) to critical systems (e.g., finance, healthcare, e-governance, and e-commerce) (Aggarwal et al., 2019; Hewa et al., 2021). However, the lack of interoperability between heterogeneous blockchain systems is one of the biggest challenges nowadays. Over the last few years, a number of solutions in this research direction have been proposed in the literature (Belchior et al., 2021). Among them, one of the promising solution is the pub-sub interoperability protocol proposed by the Linux Foundation (Ghaemi et al., 2021), which introduces a blockchain-backed broker/dispatcher responsible for message communication among various publishers and subscribers' blockchain systems in the many-to-many setting. This enhances the scalability of the overall system and makes this solution profitable.

Let us consider a critical scenario where cross-border inter-bank payments and settlements occur among multi-stakeholders participating in various supply chains and trade finances. This involves an interaction between the stakeholders from different

heterogeneous blockchain platforms that assimilates the pub-sub blockchain interoperability protocol for intercommunication. To exemplify it, we assume a stakeholder A, who is already registered in a central bank digital currency blockchain network X of a country, wishes to join a supply chain network Y. In this case, A's know your customer (KYC) details, which are already verified by X, can be used to authenticate her in the network Y, allowing her to join Y and to perform financial trade in Y and payment settlement via X. It is apparent that, in such situations, any fault in the system (e.g., message loss due to a smart contract's functional error) may lead to substantial monetary loss. Thus, there is a dire need to verify the robustness, requirements, and design of such a real-time critical system.

In the realm of formal verification, model checking (Clarke et al., 1994) appears as a promising algorithmic approach to reduce the burden on an expert's intervention and makes it easier to detect and fix bugs in the design process. Given a model $\mathbb{M}$ and a property ϕ, where $\mathbb{M}$ represents a system as a state-transition diagram and ϕ is expressed in some mathematical logic, the model checking algorithm performs an exhaustive case analysis on the set of states and determines whether $\mathbb{M}$ satisfies ϕ, i.e., $\mathbb{M} \models \phi$.

While there have been a number of proposals on formal verification of various properties confined to individual blockchain platforms, such as Bitcoin protocols (Chaudhary et al., 2015; Chaudhary et al., 2020; DiGiacomo-Castillo et al., 2020), Ethereum smart contracts (Abdellatif and Brousmiche, 2018; Nehai et al., 2018; Osterland and Rose, 2020), and Hyperledger Fabric chaincodes (Alqahtani et al., 2020; Liu et al., 2022), researchers have not paid attention to formally verifying the correctness of the blockchain interoperability protocol as stated in Tolmach et al. (2021). In this paper, we propose a systematic approach to model and formally verify the real-time properties of the pub-sub interoperability protocol, with a special focus on the message communication through API calls among publishers, subscribers, and brokers. This is worthwhile to mention that, even though this protocol can be adopted by any decentralized applications, we aim to verify only the protocol itself, instead of any particular application adopting the protocol. In this direction, we build an abstract stochastic timed automata (STA) model from the source code of the protocol by extracting relevant information guided by our properties of interest. In addition to the verification of real-time properties, we also provide a probabilistic estimation of various functionalities involved in this complex protocol. To the best of our knowledge, this is the first proposal of its kind to model and verify the blockchain pub-sub interoperability protocol using model checking.

To summarize, our main contributions in this paper are

- Formalization of blockchain smart contracts (chaincodes) in the form of finite-state transition systems.
- Modeling of the pub-sub interoperability protocol as stochastic timed automata, describing its construction from the source codes by analyzing and extracting the call graph and contextual information.
- Real-time property verification of the pub-sub interoperability protocol using the UPPAAL-Statistical Model Checker (UPPAAL-SMC).
- Development of a proof-of-concept Chaincode Analyzer that generates call graphs from JavaScript chaincodes and extracts relevant contextual information from each function. The extracted information and the call graph are then used for modeling the system as a network of STA in UPPAAL-SMC using the drag and drop feature.
- Performance evaluation of the Chaincode Analyzer on a set of representative chaincodes collected from the pub-sub protocol and Hyperledger official GitHub repository and probabilistic estimation of various functionalities of the protocol by varying the throughput of the different publishers and subscribers.

The rest of the paper is structured as follows: Section 2 gives an overview of related work. Section 3 recalls the basic background knowledge on timed automaton, stochastic timed automaton, and UPPAAL-SMC. Section 4briefly describes the pub-sub blockchain interoperability protocol proposed by the Linux Foundation. Section 5 provides the details of our proposed modeling approach. Section 6 discusses the details of the proof of concept. Section 7.1 reports the performance evaluation of the *Chaincode Analyzer*. The verification results of the protocol with respect to the properties of interest are shown in Section 7.2. Section 8 presents threats to validity. Finally, Section 9 concludes this paper.

2 Related work

Although there is no study on blockchain interoperability verification, there is considerable literature on blockchain and smart contract verification. These verifications effectively adopt the model checking approach. The Bitcoin protocol's UPPAAL model is illustrated in Chaudhary et al. (2015) to probe the double-spending attack in a situation with malicious peers. Here, the authors show the probability analysis of the double-spending attack based on the number of confirmations. In a similar line, the analysis of the double-spending attack based on the hash rate is done in Chaudhary et al. (2020). Fehnker and Chaudhary (2018) simulated a Bitcoin majority attack in UPPAAL and concluded that for a share of 20%, the attack will be effective within a few days. DiGiacomo-Castillo et al. (2020) used UPPAAL-SMC to investigate Bitcoin backbone protocol features that vary as a function of concrete parameters in a network where an attacker can undertake selfish mining. Similarly, Andrychowicz et al. (2014) proposed a framework to model the Bitcoin contracts employing the timed automata in the UPPAAL. Eijkel and Fehnker (2019) modeled the behavior of honest and selfish mining pools in UPPAAL. It includes network delay but, unlike earlier models in the literature, does not presume a single view of the blockchain.

Abdellatif and Brousmiche (2018) modeled Solidity smart contract and blockchain execution protocol along with users' behaviors to analyze the design vulnerabilities of smart contracts using a statistical model checking tool. A tool chain is developed in Osterland and Rose (2020) for translating Solidity smart contracts to generate an automata-based code representation in PROMELA, which is verified by the SPIN model checker. Similarly, in Bai et al. (2018), smart contracts are modeled in PROMELA, and the SPIN model checker is used to ensure that a contract's logic is valid. In Nehai et al. (2018), the behavior of Ethereum blockchain, smart

TABLE 1 A comparative summary of the existing literature.

Proposals	Problem under consideration	Blockchain platform	Language under consideration	Properties under verification	Specification language	Tools used
Andrychowicz et al. (2014)	Bitcoin contracts	Bitcoin	Bitcoin script	Safety properties	TCTL	UPPAAL
Chaudhary et al. (2015)	Double spending in Bitcoin	Bitcoin	Bitcoin script	Probability of double spending based on the depth of the block	MITL	UPPAAL-SMC
Fehnker and Chaudhary, (2018)	Majority attack analysis and optimization proposed by Bitcoin Unlimited	Bitcoin Unlimited	Bitcoin script	Time analysis for an attack based on confirmation depth	MITL	UPPAAL-SMC
Abdellatif and Brousmiche, (2018)	Ethereum smart contracts	Ethereum	Solidity	Safety along with functional properties	PB-LTL	BIP-SMC
Bai et al. (2018)	Smart contracts	Generic	Generic	(i) State accessibility (ii) No deadlock (iii) No livelock	LTL	SPIN
Nehai et al. (2018)	Ethereum smart contracts	Ethereum	Solidity	Functional behavioral properties	CTL	NuSMV
Atzei et al. (2018)	Bitcoin transactions	Bitcoin	Bitcoin script	(i) Double spending transactions (ii) Overall value contained in blockchain	Balzac	Balzac
Eijkel and Fehnker, (2019)	Selfish mining in Bitcoin	Bitcoin	Bitcoin script	Effects of selfish mining	TCTL and MITL	UPPAAL-SMC
Mavridou et al. (2019)	Ethereum smart contracts	Ethereum	Solidity	(i) Safety (ii) Liveness (iii) Deadlock freedom	N/A	BIP and NuSMV
Chaudhary et al. (2020)	51% attack on Bitcoin network	Bitcoin	Bitcoin script	Probability of majority attack varying hash rate	MITL	UPPAAL-SMC
DiGiacomo-Castillo et al. (2020)	Bitcoin backbone protocol	Bitcoin	Bitcoin script	(i) Chain quality (ii) Common prefix (iii) Chain growth	MITL	UPPAAL-SMC
Osterland and Rose, (2020)	Ethereum smart contracts	Ethereum	Solidity	(i) Assertions (ii) Deadlock detection (iii) Liveness properties	LTL	SPIN
Alqahtani et al. (2020)	Chaincodes for supply Chain management system	Hyperledger Fabric	Go	Behavior correctness and functional requirements	LTL	NuSMV
Zhang et al. (2020)	CKB block synchronization	Nervos Common Knowledge Base	N/A	Correctness and consistency of the protocol	TCTL and MITL	UPPAAL-SMC
Gu et al. (2022)	Raft and PRaft consensus	N/A	N/A	(i) Single leader (ii) Leader completeness (iii) Single leader (iv) Leader completeness	TLA+	TLC
Bertrand et al. (2022)	DBFT Consensus	Redbelly blockchain system	N/A	(i) Safety (ii) Liveness	LTL	ByMC
Afzaal et al. (2022)	Blockchain-based crowdsourcing	N/A	N/A	Safety, fault tolerance, trusted leader, and trusted validator properties	LTL	PAT

(Continued on following page)

TABLE 1 (*Continued*) **A comparative summary of the existing literature.**

Proposals	Problem under consideration	Blockchain platform	Language under consideration	Properties under verification	Specification language	Tools used
Liu et al. (2022)	Port supply chain	Hyperledger Fabric	N/A	Safety, reliability, and reachability	PCTL	PRISM
Park et al. (2022)	Dutch auction trading system	N/A	N/A	Basic principles and functional requirements	TCTL	UPPAAL
Nam and Kil, (2022)	Ethereum smart contracts	Ethereum	Solidity	Functional behavioral properties	ATL	MCMAS
Our Proposal	**Pub-Sub Interoperability**	**Hyperledger Fabric**	**JavaScript**	**(i) Reachability**	**TCTL and MITL**	**UPPAAL-SMC**
				(ii) Liveness		
				(iii) Probability estimation of message interoperability based properties		

contracts, and the execution framework are captured by the three layers of the model, respectively, and the properties of smart contracts expressed in computation tree logic (CTL) are verified by using the model checker NuSMV. Alqahtani et al. (2020) used the NuSMV model checker to model the Hyperledger Fabric smart contracts and their interactions with the aim of verifying their compliance with the systems' functional requirements.

Mavridou et al. (2019) introduced the VeriSolid framework to generate the Solidity code from the verified transition system-based models following the correct-by-design development principle. Bartoletti and Zunino (2019) compared five formal modeling techniques, namely, Balzac, Ivy, Simplicity, UPPAAL, and BitML, for Bitcoin smart contracts based on their expressiveness, usability, and suitability for verification. Atzei et al. (2018) proposed a formal model of Bitcoin transactions which is abstract enough to allow for formal reasoning of the behavior of Bitcoin transactions. However, as highlighted in Atzei et al. (2018), there are some differences between the modeling of Bitcoin scripting language and the blockchain with respect to the actual ones. This is worthwhile to mention that Palina Tolmach et al. in their survey (Tolmach et al., 2021) revealed that reasoning about the functional correctness of smart contracts across all domains is frequently done using a combination of contract-level models, specifications, and model checking.

Gu et al. (2022) used the interactive preserving abstraction (IPA) framework to verify the performance of two blockchain consensus protocols Raft and PRaft with compositional model checking using TLA + language. The set of properties verified are as follows: single leader and leader completeness, singleLeader, and leaderCompleteness blockchain. Bertrand et al. (2022) holistically verified the safety and liveness properties of the Democratic Byzantine Fault Tolerant consensus used in the Redbelly Blockchain system, a scalable blockchain used in production. Afzaal et al. (2022) presented blockchain-based crowdsourcing consensus protocol formal models built using CSP# and verified using the PAT model checker. Liu et al. (2022) modeled the Hyperledger Fabric chaincodes for the port supply chain system as discrete-time Markov chains and validate the properties in PCTL (Probabilistic Computation Tree Logic) using the PRISM model checker. The UPPAAL model checker is used in Zhang et al. (2020) and Park et al. (2022) to formally verify the public descending auction system (Dutch Auction) and important properties of the Common Knowledge Base (CKB) block synchronization protocol, respectively. Nam and Kil (2022) proposed the formal verification of Solidity smart contracts using the ATL model checker.

A comparative summary of the existing relevant verification approaches with respect to our proposal is shown in Table 1, where MITL stands for metric interval temporal logic, TCTL stands for timed computation tree logic, PB-LTL stands for probabilistic bounded linear temporal logic, LTL stands for linear temporal logic, PCTL stands for probabilistic computation tree logic, CTL stands for computation tree logic, and ATL stands for alternating-time temporal logic.

3 Timed automata and stochastic timed automata

Timed automaton (Bengtsson and Yi, 2004) is used worldwide as a powerful model to specify and verify real-time systems. It is basically a finite-state machine (FSM) extended with a finite set of real-valued clocks. Initially, all the clocks of the system are set to 0 and increase synchronously as time passes by. The individual clocks can be reset to 0 depending upon certain transitions taken in the system. Here, the transitions are labeled by constraints imposed over clock variables (also known as clock-constraints) to limit the behavior of an automaton. Timed automaton over a finite set of atomic propositions is formally defined as follows:

Definition 1 (Guard). *A Guard is a finite conjunction of expressions of the form* $x \oplus c$ *or* $x - y \oplus c$, *where* $x, y \in X$ *are clocks,* $c \in \mathbb{N}$ *is a number, and* $\oplus \in \{\geq, >, =, <, \leq\}$

Definition 2 (Timed Automaton). *A Timed Automaton T is a tuple* $\langle L, l_0, \Sigma, X, E, \mathcal{I} \rangle$, *where (i) L is a finite set of locations, (ii)* $l_0 \in L$ *is the initial location, (iii)* $\Sigma = \Sigma_i \cup \Sigma_o$ *is the alphabet of actions partitioned into input*(Σ_i) *and output*(Σ_o), *(iv) X is a finite set of clocks, and (v) E is a finite set of transitions edge such that* $E \subseteq L \times \mathcal{GRD}(X) \times \Sigma \times 2^X \times L$, *where* $\mathcal{GRD}(X)$ *is a guard over X, and (vi)* $\mathcal{I}: L \rightarrow \mathcal{GRD}(X)$ *assigns an invariant (which could be empty) to each location.*

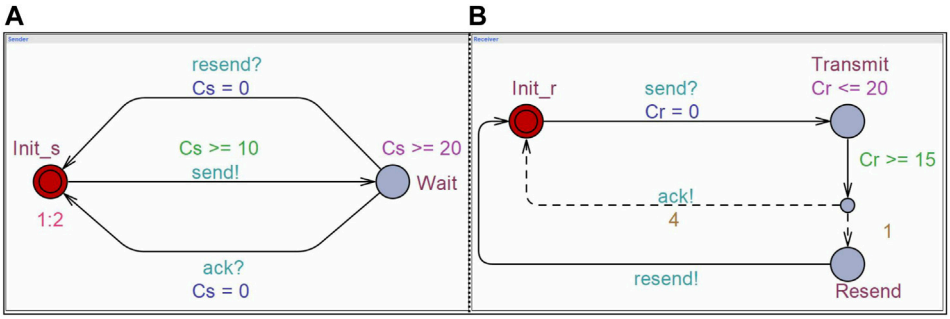

FIGURE 1
Network of two stochastic timed automata, modeling a sender **(A)** and a receiver **(B)**.

Intuitively, a transition in a location ℓ can evolve either by (1) *delay transition*: remaining in the current state by letting time pass, i.e., incrementing all clocks, provided the invariant $\mathcal{I}(\ell)$ can continuously be satisfied, or (2) *action transition*: making a transition (ℓ, g, a, C, ℓ') if the conditions g and $\mathcal{I}(\ell')$ hold, going to the location ℓ' and setting the clocks in C to 0.

A stochastic timed automaton (STA) (Cassandras and Lafortune, 1999) is a timed automaton which incorporates a stochastic clock structure, an initial state cumulative distribution function (CDF), and state transition probabilities. The STA is formally defined as follows.

Definition 3 (STA). *An STA S_{TA} is a tuple $\langle \xi, L, \Gamma, p, p_0, G \rangle$, where (i) ξ is a set of finite-state transitions (also known as. event set), (ii) L is a finite set of locations (i.e., state space), (iii) $\Gamma(\ell)$ is a set of enabled or feasible transitions defined for all $\ell \in L$ with $\Gamma(\ell) \subseteq \xi$, (iv) $p(\ell'; \ell, e')$ is state transition probability defined for all ℓ', $\ell \in L$; $e' \in \xi$, and such that for all $e' \notin \Gamma(\ell)$, $p(\ell'; \ell, e') = 0$, (v) $p_0(\ell)$ is the probability mass function $P[L_0 = \ell]$, $\ell \in L$, of the initial state L_0, and (vi) $G = \{G_i : i \in \xi\}$ is a stochastic clock structure, where G_i is a distribution function for each event i, describing the random clock sequence.*

A set of STA running simultaneously, using the same set of clocks, and communicating via broadcast channels and shared variables are called *Network of Stochastic Timed Automaton* (NSTA). The NSTA is supported by the UPPAAL-SMC tool, where the communication is limited to broadcast synchronization. As UPPAAL-SMC deals with probabilistic property verification, this is a design choice by them in order to capture a clean semantics of only non-blocked components which are racing against each other with their corresponding local distributions. We follow the convention for broadcast synchronization action as a! and a?, where these notations mean performing a! triggers a? to be performed (David et al., 2015). We assume that the NSTAs are *input enabled, deterministic*, and *non-zero* (i.e., time always diverges). Furthermore, we drop clock constraints (if true), actions (if irrelevant), and reset sets (if ϕ) from the labels. Let us understand this with an example of a network of two STAs.

Figure 1 illustrates a network of two STAs portraying communication between a sender and a receiver. When the receiver successfully receives a message, it sends an acknowledgement with some probability. Otherwise, it asks the sender to resend the message. We use different colors to distinguish among distinct elements, such as *node invariant, guards, synchronizations, updates, delay transition rate, and discrete transition rate*, of the automata. The dotted lines denote probabilistic transition. Initially, the sender S and the receiver R both remain in initial locations Init_s, Init_r, respectively, and their clocks Cs and Cr are set to 0. The time-delay transition rate 1:2 associated with the location Init_s indicates that the non-deterministic choices of time-delay transition are chosen according to exponential distribution with user-defined rate 1:2 (in the absence of invariant). S waits as per the delay rate and then triggers send! to move to Wait state, thus making R to perform send? and to move to Transmit state. R waits in this state for at most 20 time units before it can fire ack! or resend! with a probability of 4/5 or 1/5, respectively. S now waits at least 20 time units before listening to fired channel and then moves back to the initial position by resetting the clock Cs to zero. The guards used for the sender and receiver induce a delay of 10 and 15 time units, respectively, while taking a transition.

3.1 Property verification using UPPAAL-SMC

UPPAAL-SMC (David et al., 2015) is an extension to the UPPAAL model-checker toolbox, based on the theory of STA for analysis of probabilistic performance properties. UPPAAL-SMC espouses property representation using the following temporal logics: (a) *TCTL* for specifying desired properties over the network of timed automata and (b) *MITL* for specifying desired properties over the network of STA. Let ϕ be a path formula that uses a state formula ψ defined over the clocks and the locations of automata. The syntax of TCTL and MITL in BNF is defined as follows.

Definition 4 (TCTL formulas). *The temporal logic TCTL is defined by the following grammar:*

$$\phi := \exists \Diamond \psi \mid \exists \Box \psi \mid \forall \Diamond \psi \mid \forall \Box \psi \mid \psi_1 \rightarrow \psi_2$$
$$\psi := T.s \mid CC \mid \neg \psi \mid \psi_1 \vee \psi_2 \mid \psi_1 \wedge \psi_2 \mid \psi_1 \Rightarrow \psi_2$$

where $T.s$ denotes state s in the timed automaton T; CC is a clock constrain, the operators $\neg, \vee, \wedge$ and $\Rightarrow$ are logical negation, disjunction, conjunction, and implies, respectively; the operators $\exists$ (*There exists*), $\forall$ (*For all*), $\Box$ (*Globally*) and $\Diamond$ (*Future*) are temporal operators describing state's range for which ψ must hold; and $\psi_1 \rightarrow \psi_2$ denotes ψ_1 leads to ψ_2 defined as $\forall \Box (\psi_1 \Rightarrow \forall \Diamond \psi_2)$.

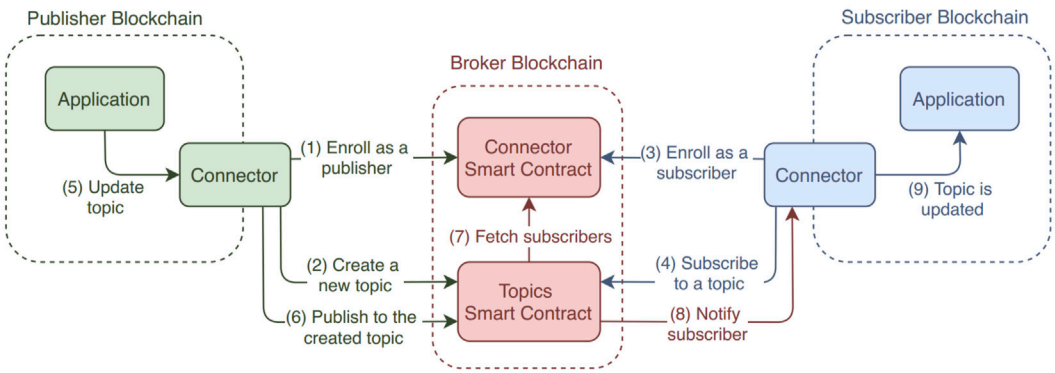

FIGURE 2
Architecture of the protocol and message flow (Ghaemi et al., 2021).

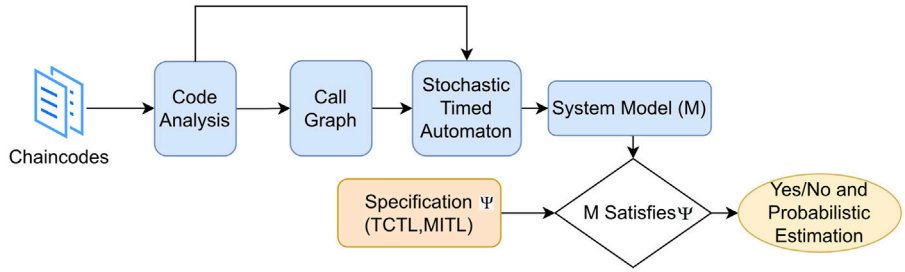

FIGURE 3
Generic approach for modeling and verification of chaincodes in the pub-sub interoperability protocol.

Definition 5 (MITL formulas). *The temporal logic MITL with weighted extension is defined by the following grammar:*

$\varphi := AP \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \bigcirc\varphi \mid \varphi_1 \sqcup^{c}_{\leqslant n} \varphi_2$

where AP is a conjunction of atomic propositions over the state of the STA, and $\bigcirc$ is the next state operator. The MITL formula $\varphi_1 \sqcup^{c}_{\leqslant n} \varphi_2$ is satisfied if φ_1 is satisfied until φ_2 is satisfied in the clock time limit of n for clock c.

The temporal operators of TCTL can be used to specify properties under the following categories (Pnueli, 1977; Behrmann et al., 2004).

- *reachability:* This class refers to the properties, which answers the question "is it possible to end up in a given state?" Usually, they are expressed using the path formula $\exists\Diamond\varphi$.
- *safety:* This class of properties ensures that something bad never happens. We express that φ should be true in all reachable states with the path formula $\forall[]\varphi$.
- *liveness:* This class of properties ensures something good will eventually happen. Usually, these are stated as $\forall\Diamond\varphi$ and $\varphi \rightarrow \psi$.
- *fairness:* It refers to the properties which address the question "does, under certain conditions, an event occur repeatedly?"
- *functional correctness:* This class refers to the properties that answer the query "does the system do what it is supposed to do?"
- *real-time properties:* This class refers to the properties which acknowledge the question "is the system acting in time?"

To exemplify, let us express in TCTL two properties of the network of STA in Figure 1: $\exists\Diamond$ S. Wait $\Rightarrow$(Cs < 20), which states that the clock Cs can be less than 20 time units while in location Wait in S. Similarly, the TCTL formula S.Wait −−> R.Transmit represents that the location R.Transmit is always reachable if the location S.Wait is reached.

Given a clock x and a bound C, the statistical algorithms in UPPAAL-SMC use MITL formulas to answer the following three categories of queries: (a) *probability estimation*: it is expressed as $\mathbb{P}_M(\Diamond_{x\leqslant C} AP)$, which answers the following question "what is the probability of the network of Stochastic Timed Automata M satisfying property AP within C time limit?" (b) *hypothesis testing*: this answers the query "Is the probability $\mathbb{P}_M(\Diamond_{x\leqslant C} AP)$ of the network of Stochastic Timed Automata M greater or lesser than a specific threshold?" (c) *probability comparison*: it addresses the question "Is the probability $\mathbb{P}_M(\Diamond_{x\leqslant C} AP_1)$ greater than the probability $\mathbb{P}_M(\Diamond_{y\leqslant D} AP_2)$?" An example property in MITL is: Pr[< = 100] ($\Diamond$ R.Resend), which estimates the probability that receiver R will be requesting to resend the message within 100 time-units.

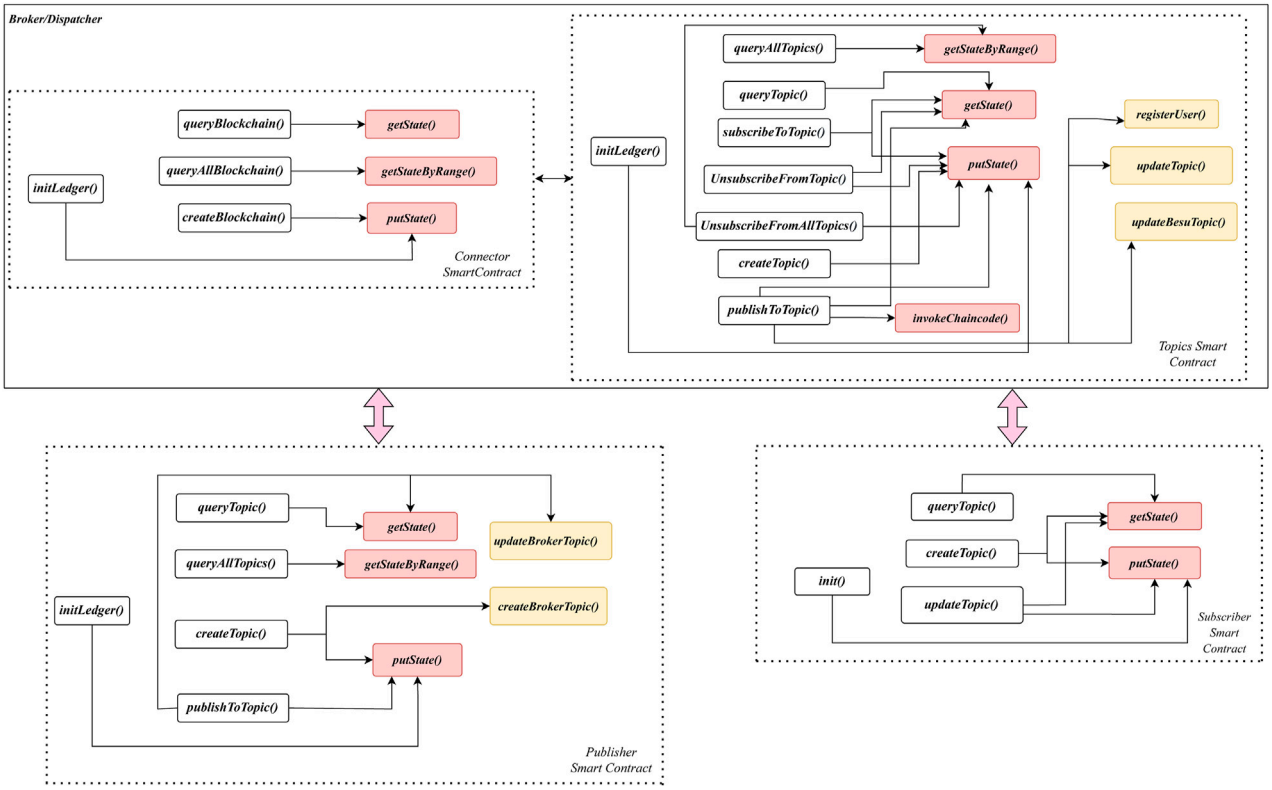

FIGURE 4
Call graph of the pub-sub interoperability protocol.

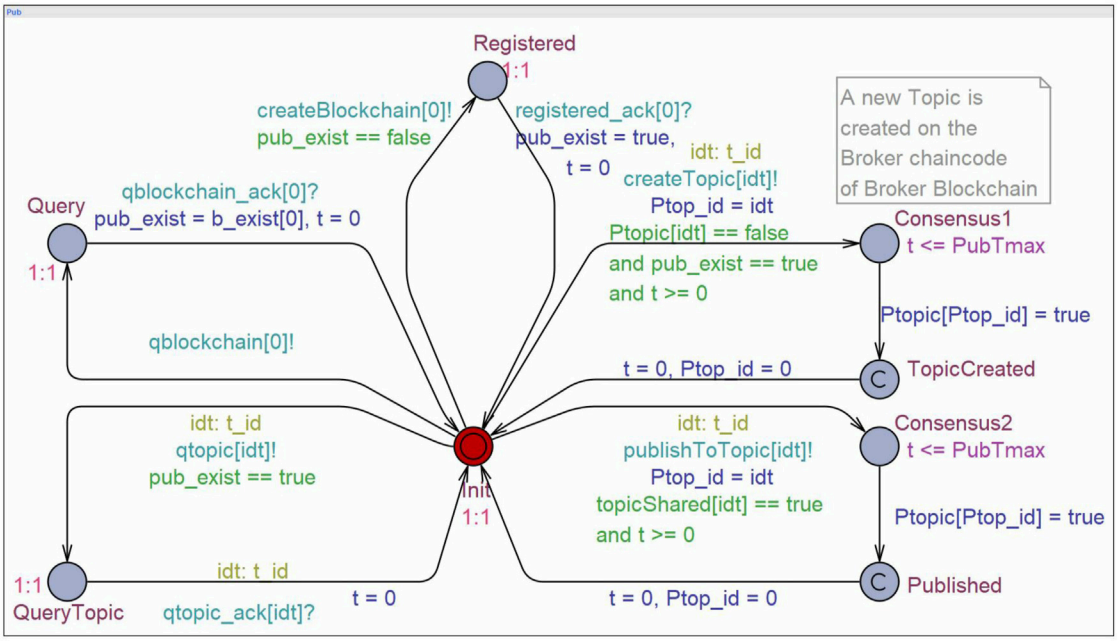

FIGURE 5
Publisher template.

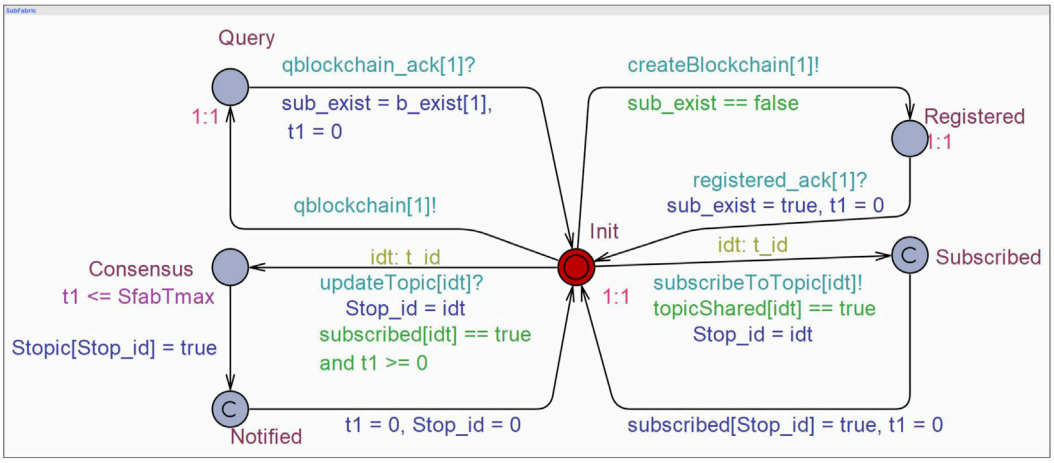

FIGURE 6
Subscriber template.

4 Publisher–subscriber-based blockchain interoperability protocol

Given the ever-growing demand to enhance inter-blockchain communication, a substantial amount of research effort has been given over the last couple of years. Among them, a notable solution is proposed by the Linux Foundation, which adopts a publish–subscribe architecture to establish interoperability, as depicted in Figure 2. The interoperability among a number of heterogeneous blockchain platforms, falling under the Hyperledger umbrella project (Foundation, 2015), is established by implementing a number of chaincodes in JavaScript language with the required functionalities and API calls.

Hyperledger (Foundation, 2015) is an umbrella project with multiple platform open-source collaborative efforts anchored by the Linux Foundation. Few of the popular projects include Hyperledger Fabric, Hyperledger Besu, Hyperledger Aries, and Hyperledger Sawtooth, where each focuses on solving distinct classes of problems. Among them, Hyperledger Fabric is one of the most popular development platforms for distributed ledger solutions adopted by the business enterprises. The business logic in Hyperledger Fabric is encoded using smart contracts, also known as chaincodes, which support generic programming languages such as Java, Go, and JavaScript. Chaincode is a program that initializes and oversees the ledger states in a blockchain network with the help of transactions submitted by applications. In general, a chaincode can be invoked by end-users either to update or to query the ledger via a transaction.

Let us now briefly manifest the main components of the pub-sub interoperability protocol and the flow of information among them as follows.

1. *Publisher blockchain:* This is the source blockchain that participates in the network by implementing an appropriate connector chaincode to interact with the broker's blockchain chaincodes. It enrolls as a publisher to the broker's connector chaincode and is responsible for creating and publishing messages on as many topics as required.
2. *Subscriber blockchain:* This is the destination blockchain that participates in the protocol similar to that of the publisher. It enrolls as a subscriber to the broker's connector chaincode and receives the messages whenever a publisher updates the topic to which it is subscribed through the broker's topic chaincode.
3. *Broker blockchain:* This is the critical component of the architecture responsible for establishing the interoperability of messages from publishers to subscribers. It has two chaincodes: *connector* chaincode, which is accountable for maintaining all details of the publishers and subscribers in the network, and *topic* chaincode, which is responsible for maintaining the details of the topics, i.e., their publishers, subscribers, and messages.

Every blockchain joining the protocol can act as either a *publisher* or *subscriber*. Publishers can create *n* number of topics, and a single topic can have *m* number of subscribers. This pub-sub architecture provides primarily six functionalities to ensure the interoperability among the heterogeneous blockchain platforms. These functionalities are discussed as follows.

- *Register/Enroll:* The publisher/subscriber implements the appropriate connector chaincode for this functionality and then enrolls to the broker network via the broker's connector chaincode. These are shown in steps (1) and (3) in Figure 2.
- *Create_Topic:* The publisher, after getting registered, may create a new topic to which a registered subscriber can subscribe via the broker's topics chaincode, as shown in step (2).
- *Subscribe_to_Topic:* The connector chaincode of a registered subscriber offers this functionality to subscribe to any new topic created by a publisher via the broker blockchain, as shown in step (4).
- *Update_Topic and Publish_to_Topic:* When updates happen in the publisher network, the message gets published to the topic already created by the publisher. The publish-request is

then sent to the broker's topics chaincode, which fetches the details of the subscribers of the topic from the connector chaincode. These are depicted in steps (5), (6), and (7).

- *Notify:* Finally, the broker notifies all the subscribers of a topic about the updated message, which the subscribers update in their networks as shown in step (8).

5 Proposed methodology

There have been considerable research studies dedicated to blockchain interoperability in recent years, resulting in the emergence of several protocols. The pub-sub interoperability protocol proposed by the Linux Foundation provides a potential solution to establish inter-blockchain communication among different blockchain platforms under the umbrella of Hyperledger projects. This makes the protocol generic enough to be adopted by various decentralized applications (e.g., supply chain, healthcare, trade finance, and cross-border payments) built on the top of Hyperledger platforms, where one can act as a publisher and other can act as a subscriber to establish communication among them. In this section, our main objective is to systematically model the protocol and to formally verify its real-time properties, instead of the properties of any particular application adopting it. More specifically, various decentralized applications which adopt the pub-sub interoperability protocol to establish inter-blockchain communication among them generally consist of the following two classes of chaincodes: (1) *application-specific chaincodes,* which perform application-specific tasks and (2) *protocol-specific chaincodes,* which perform pub-sub protocol-specific tasks. As our properties of interest deal with the correctness of only protocol-specific tasks, our verification approach aims at modeling and verification of protocol-specific chaincodes, instead of the application-specific chaincodes. Figure 3 depicts the overall modeling and verification architecture. Observe that, as the protocol-specific functionalities focus on the message communication through API calls among publishers, subscribers, and brokers, they are independent of the application-specific activities. This makes our proposed approach applicable to any protocol-specific chaincodes, irrespective of the applications adopting the protocol. Let us now explain each of these phases in detail.

5.1 Formalizing chaincode

In this subsection, we formally model chaincodes written in JavaScript as an FSM. Our theoretical formalization serves as a foundation to model any JavaScript chaincode in the form of a state transition system guided by the properties of interest and the verification thereof. Here, we consider a subset of JavaScript language statements, consisting of *Identifiers, Types, Supported Statements, Variables,* etc., used in the chaincode. We portray the chaincode as an FSM according to the following rules (Mavridou et al., 2019).

1. A function is mapped as a transition where the action associated with the transition corresponds to the set of statements in the function's scope.
2. When a transition takes place, the corresponding statements representing the action are executed.
3. A guard is a premise on variables that must be true to allow the associated transition.

Definition 6 provides a formal definition of a chaincode in the form of the FSM.

Definition 6. *A chaincode* **C** *is defined as a tuple* $\mathbf{C} = (S, s_0, F, \Sigma, a, \delta)$, *where (1)* S *is a finite set of states which is a subset of valid identifiers used in the chaincode; (2)* s_0 *is the initial state, where* $s_0 \in S$; *(3)* F *is a set of final states, where* $F \subseteq S$; *(4)* Σ *is a set of contract's typed variables defined as* $\Sigma \subset$ *Identifiers* $\times$ *Types; (5)* a *is an action, where* $a \in$ *Supported Statements; and (6)* δ *is a transition relation, where for each* $t \in \delta$, *there exists the following: (i) transition name id* $\in$ *Identifiers, (ii) source state* $s_{from} \in S$, *(iii) destination state* $s_{to} \in S$, *(iv) arguments* arg $\subset \Sigma$, *(v) transition guard* g, *(vi) return variables* $r_t \subset \Sigma$, *and (vii) action* $a_t \subset a$.

This definition serves as a basis to model the chaincodes of the pub-sub interoperability protocol as an NSTA, in line with Definition 3.

5.2 Chaincode analysis

Chaincodes written in JavaScript consist of a number of functions encoding business logic which, when executed, changes the state of the blockchain. More precisely, as our objective is to prove real-time properties about the successful creation/subscription of topics by publishers/subscribers, seamless message transmission between entities, successful registration to broker network, etc., the JavaScript chaincode analyzer extracts relevant information from the protocol source codes with respect to these properties of interest. This includes unique identities of various entities (e.g., topics, publishers, subscribers, and brokers), built-in functions changing the blockchain state, context-sensitive information pertaining to various function calls, etc. This allows the construction of a *call graph* by identifying all possible function calls among various chaincodes involved in publishers, brokers, and subscribers, along with the associated contextual information. This provides essential information about the inter-procedural control flow dependencies in the pub-sub protocol. Figure 4 illustrates the call graph of the pub-sub interoperability protocol. The functions responsible for API calls or facilitating the API calls are represented with two different colors in the call graph. In particular, the blockchain-specific API calls accountable for updating the world state and ledger of the network are shown in pink, whereas the API calls which are responsible for interoperability are shown in yellow.

5.3 Modeling of the pub-sub interoperability protocol

To perform model checking of a system, an essential requirement is the representation of the system in the form of a model. Based on the extracted information by the analyzer, we build a model of the protocol in the form of an NSTA using UPPAAL-SMC's drag and drop features in a systematic manner by defining the states, transitions, guards, invariants, updates, and

synchronization channels. This guided approach enables us to build an abstract model of the protocol reflecting similar behavior as of the source code, disregarding the details which are not relevant with respect to the targeted properties of interest. In this process, we make a number of assumptions, without loss of generality, which consider hardware dependencies, fairness in channel behavior, absence of issues stemming from hardware components, etc. Moreover, since the Hyperledger blockchain networks are matured, we assume the network to be safe and robust, thus abstracting the block generation process and consensus algorithm details in our proposed model.

Let us discuss the model generation steps in detail. First, based on the generated call graph by the analyzer as shown in Figure 4, we identify a number of synchronization channels required to communicate among STA in the network. In particular, the synchronization channels are mapped from the functions responsible for external function calls. The locations in the automata are identified during the code analysis phase, which corresponds to the state change occurring in the pub-sub interoperability protocol. The state transitions are mapped from the call graph by identifying the apt function responsible for the state change. The global variables infused in the model are extracted during chaincode analysis to facilitate the data exchange among the NSTA. Along with this, the *guard*, *conditional checks*, and *updates* are introduced after analyzing the algorithms used in the chaincodes. Finally, a clock, an important component related to time, is introduced to model the throughput/latency of the blockchain network.

We consider throughput as a crucial parameter during the modeling of the blockchain system. Let t_{x_i} be the processing time of the *ith* transaction, t_{b_i} be the *ith* block commit time in a blockchain network, and N is the total number of transactions per block. The *throughput* (in transaction per second (TPS)) and *latency* (in milliseconds (ms)) of the network are defined as follows (Dreyer et al., 2020):

$$\text{Throughput} = N \Big/ \sum_{i}^{N} t_{x_i}, \tag{1}$$

$$\text{Latency}_i = t_{x_i} + t_{b_i}. \tag{2}$$

Every blockchain platform has a specific throughput value indicating the number of successful TPS. Given the throughput value, we calculate the average time (in seconds) taken by each successful transaction. The clocks in the model keep track of this calculated time, which is used as an invariant to reflect the consensus occurring in the blockchain network. For example, the publisher blockchain (i.e., Fabric V2) throughput value is 20,000 TPS which implies the average time for a successful transaction is 50 microseconds, which is incorporated in the *consensus* state of the model. In general, the real behavior of the consensus time is reflected by latency. However, due to its dependence on hardware, network delay, transaction processing fees, miners, etc., we assume that the block commit time t_{b_i} is negligible in Eq. 2 and thus, we use only the throughput parameter for modeling the consensus time[1] as discussed previously. After modeling the template, it can be transformed into an NSTA.

The pub-sub-based solution architecture, described in Section 4, is classified into four actors and objects: publisher (creator of content), subscriber (consumer of content), broker blockchain (keeping track of publisher and subscriber blockchains), and broker topics (keeping track of topics involved in message communication). Now, we are in a position to discuss the detailed specification of these actors, represented in the form of STA models, which are verified against several properties using the UPPAAL-SMC model checker[2]. Figures 5–8 depict various templates modeled in UPPAAL-SMC, which are elaborated upon as follows.

5.3.1 Publisher

The primary task of a publisher after enrolling to the pub-sub interoperability network is to create a new topic and publish messages over the topic. Furthermore, a publisher can query about topics created by it and whether it has already been registered. Accordingly, we identify six unique states ***Init***, ***Query***, ***QueryTopic***, ***Registered***, ***TopicCreated***, and ***Published***, depicting the functionalities of the publisher. Additionally, two states, *Consensus1* and *Consensus2*, reflect the consensus occurring before a new topic is created and before a message is published to the topic, respectively. This is done in order to show the updates occurring in the world state and the ledger of the publisher network. Moreover, since there can be two categories of transactions (i.e., query and submit/invoke), we model their behavior differently. We use the notion of consensus time only for submit/invoke transactions as it changes the world state and the ledger of the blockchain.

Figure 5 shows the template model of the publisher's connector chaincode derived from the call graph in Figure 4. From the call graph, synchronization channels are identified by examining the API call functions. For example, *createTopic*() function in the call graph corresponds to the channel *createTopic [idt]*, where *idt* is the unique topic id. We append guard, update, and invariant over the transitions based on the data extracted through the chaincode analysis. The publisher from the ***Init*** state makes a transition to the ***Query*** state to know whether it is registered on the broker network. The transition is synchronized with the broker's blockchain model via channels *qblockchain* and *qblockchain_ack* parameterized with blockchain id. Here, the global array variable *b_exist* represents either enrollment (1) or non-enrollment (0) of the publisher/subscriber blockchains with a unique id. The local variable *pub_exist* copies the value from *b_exist* and thus indicates whether the publisher is registered or not.

To register on the broker network, the publisher makes a transition to the ***Registered*** state synchronized with the broker blockchain model via the channels *createBlockchain* and

1 In Hyperledger Fabric, the number of transactions per block is configurable by configtx.yaml file.

2 https://uppaal.org/

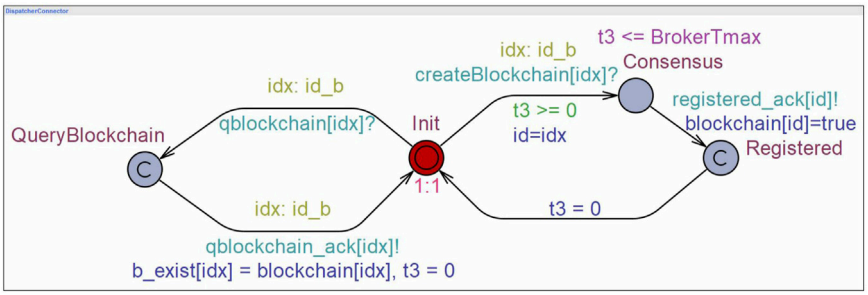

FIGURE 7
Dispatcher/broker connector template.

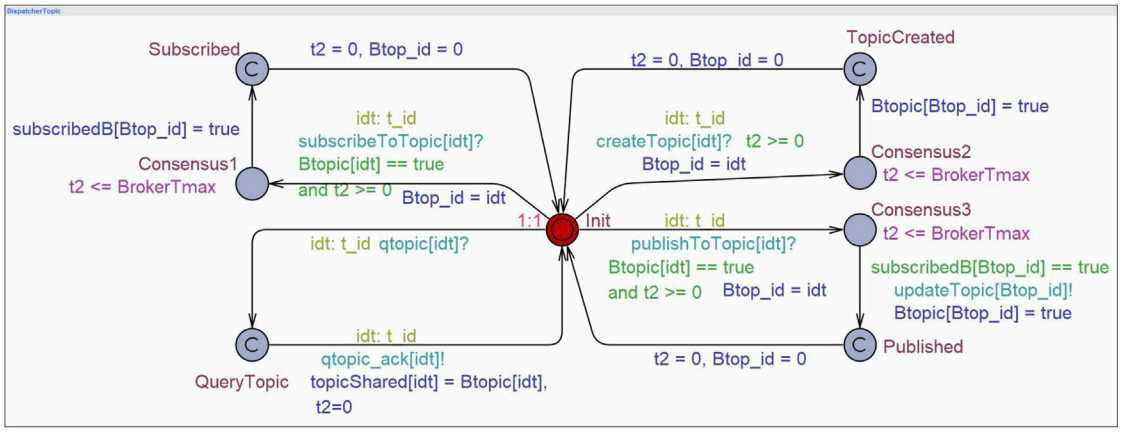

FIGURE 8
Dispatcher/broker topic template.

registered_ack. For inquiring whether a topic is already created or not, the registered publisher makes a transition to the ***QueryTopic*** state. The transition is synchronized with the broker network via channels *qtopic* and *qtopic_ack*. The local array variable of publisher *Ptopic* keeps track of all the topics created by it. So, the registered publisher for creating a new topic on both the local and broker networks makes a transition to the ***TopicCreated*** state. The transition is synchronized with the broker topic model via the channel *createTopic* parameterized with a unique topic id *t_id*. Since the topic information gets updated locally on the publisher network along with the broker network, hence intermediate *Consensus1* state is introduced at this point.

Any update in a new topic created by the publisher is notified to the subscribed subscriber. Therefore, for sending the message to the subscribers' network with the help of the broker network, the publisher makes a transition to the ***Published*** state. The transition is synchronized with the broker topic model via channel *publishToTopic* parameterized with the unique topic id *t_id*. Like before, *Consensus2* state is also infused here as the changes are reflected on the local network. In addition, the global array variable *topicShared* is used as a guard to keep track of all the topics created by the publisher. Finally, the clock *t* defined for a publisher is compared against the average time per transaction *PubTmax* value computed from the throughput of the publisher network.

5.3.2 Subscriber

Subscribers primarily perform four functionalities: (a) to query whether it has enrolled in the protocol, (b) to register itself, (c) to subscribe to a topic, and (d) to get notified if any updates occur on the subscribed topic. Accordingly, we identify four states along with an initial state. Figure 6 depicts the template model of a subscriber's connector chaincode with five unique states ***Init***, ***Query***, ***Registered***, ***Subscribed***, and ***Notified***. Similar to the publisher model, the synchronization channels in the subscriber model are identified from the call graph of the subscriber's connector chaincode shown in Figure 4.

The purpose of the transitions from ***Init*** to ***Query*** and ***Registered*** states are same as that in the case of the publisher model. To subscribe to a topic which is already created by a publisher, the subscriber makes a transition to the ***Subscribed*** state. The subscriber does this with the help of the broker network and is synchronized via the channel *subcribeToTopic* parameterized with unique topic id *t_id*. Here, the

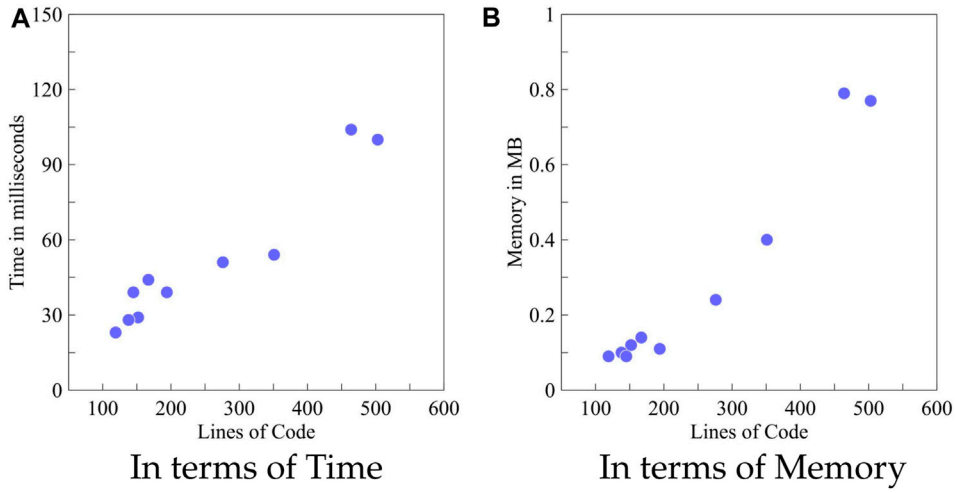

FIGURE 9
Cost of call graph construction, in terms of Time **(A)** and in terms of Memory **(B)**.

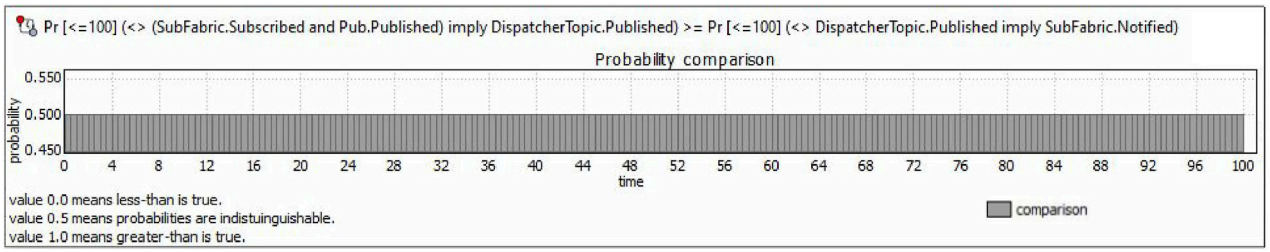

FIGURE 10
Probability comparison result.

local array variable *subscribed* keeps a record of all the subscribers for a particular topic. Moreover, when the publisher updates the data on a topic, the changes are reflected on the subscribed subscribers with the help of the broker network, and thus, to simulate this, the subscriber makes a transition to the ***Notified*** state. The transition is synchronized with the broker's topic model via the channel *updateTopic* parameterized with a unique topic id *t_id*. As the local array variable *Stopic* stores the new updated information of the topic on the subscriber network, we introduce the *Consensus* state at this point.

5.3.3 Broker connector

The broker's connector chaincode keeps a record of all the publisher and subscriber blockchains participating in the network and facilitates achieving the interoperability of message communication among them. Figure 7 shows the model of the broker's connector chaincode. It initially stays in the ***Init*** state from where it can make a transition to either (a) ***QueryBlockchain*** to provide information about all the blockchains participating in the network either as publishers or subscribers. It is synchronized with publisher and subscriber automata via channels *qblockchain* and *qblockchain_ack* parameterized with blockchain id *idx*. Here, the local array variable *blockchain* stores the data about the blockchains enrolled in the protocol, whose value is further copied to the global array variable *b_exist*, or (b) ***Registered*** state to register and store information of the blockchain which is taking part as a publisher or as a subscriber, where the broker connector is synchronized with it via array channels *createBlockchain* and *registered_ack* parameterized with blockchain id. Additionally, the clock *t3* keeps track of time, which is compared against the average time per transaction *BrokerTmax* value and is used as invariant in the *Consensus* state.

5.3.4 Broker topic

As the broker's topic automaton primarily performs four elemental functionalities, accordingly we identified four states ***QueryTopic***, ***TopicCreated***, ***Subscribed***, and ***Published*** along with an initial state ***Init***. Figure 8 depicts the template of the broker's topic chaincode, which keeps a record of all the topics created, subscribed, and updated. Similar to the publisher and subscriber models, the synchronization channels are identified. The broker from the ***Init***

state makes a transition to the ***QueryTopic*** state to inform whether a topic is already created or not on the broker network. The transition is synchronized with the publisher/subscriber via the channels *qtopic* and *qtopic_ack* parameterized with a unique topic id *t_id*. Subsequently, the local array variable *Btopic* holds the information of the topics created by the publishers and is copied to the global array variable *topicShared*.

When a publisher creates a new topic, the broker network also stores the information of this new topic and thus gets triggered to the ***TopicCreated*** state. The edge is synchronized with the publisher via the channel *createTopic*. Since the topic information gets updated locally on the broker's network along with the publisher network, hence the intermediate *Consensus2* state is infused. For storing the details of a topic subscription by a subscriber, the broker makes a transition to the ***Subscribed*** state. The transition is synchronized with the subscriber model via channel *subcribeToTopic*. The local array variable *subscribedB* used as update over the transition keeps a record of all the subscribers of a particular topic. The broker automata helps the publisher send messages to the subscriber networks by making a transition to the ***Published*** state. The transition is synchronized with the publisher via the channel *publishToTopic* and with the subscriber via the channel *updateTopic*. Since the changes are reflected on the local network, therefore *Consensus3* and *Consensus1* states are infused. In addition, the global array variable *topicShared* used as guard over the transition keeps track of all the topics created by the publisher. Finally, the clock *t2* defined for the automata is compared against the average time per transaction *BrokerTmax* value, representing the throughput of the broker network.

In all the aforementioned templates, the delay rate in the absence of the location invariant is taken as 1:1 because the delay in the original network is assumed to be uniform. However, one can change the rate to exponential distribution depending on the information available about the network *a priori*. The throughput values taken in consideration are as follows: (a) 20,000 TPS for Hyperledger fabric v2, (b) 3,000 TPS for Hyperledger fabric v1.4, and (c) 3,00 TPS for Hyperledger besu.

6 Proof of concept

The generation of a model from chaincodes involves two phases, namely, chaincode analysis and STA modeling. To accomplish the chaincode analysis phase, we have developed a proof of concept which generates call graphs of chaincodes written in JavaScript and extracts the relevant contextual information for modeling STA in UPPAAL-SMC (David et al., 2015). Our *Chaincode Analyzer* is the first of its kind to generate a call graph from a chaincode written in JavaScript. The generated call graph provides insightful information on how the procedure exchanges information among them, revealing their relationship in the program. The analysis results also include auxiliary information about the data within each procedure and global data shared among procedures.

Despite having several tools available for generating call graphs from the JavaScript code, such as code2flow (The code2flow tool, 2021), JavaScript Explorer callgraph (The javascript explorer callgraph tool, 2018), and js-callgraph (The callgraphjs tool, 2014), none of them is capable of generating a call graph from the chaincode written in JavaScript. This is due to the added complexity introduced by blockchain-based functionality that is described using special keywords in the chaincodes. In general, the existing proposals for obtaining call graphs typically builds an Abstract Syntax Tree (AST) using JavaScript compilers like Acron (Acorn, 2014) and Esprima (Esprima, 2015) However, these compilers cannot generate an AST in the case of chaincodes due to its distinct blockchain-specific features, thereby posing a unique challenge. In contrast, our proposal incorporates and implements the proven and traditional algorithm proposed by Ryder (1979) for building the call graph. Ryder's algorithm computes a precise call graph under the assumption that all call sites are invoked. One limitation of the algorithm is its inability to analyze languages that allow recursion, but since chaincodes generally avoid recursion, the algorithm is perfectly suited to our needs.

The Chaincode Analyzer performs two main functions. First, it generates an AST from the chaincodes and allows users to visualize them. Second, it generates the call graph from the given AST. We utilize the Google Closure Compiler (Bolin, 2010) to generate the AST in dot format. As the generated AST is not powerful enough to extract information regarding function calls, we implement Ryder's algorithm for generating a call graph which consists of the following two phases: initialization and construction. It contributes significantly in three distinct ways. First, the parse of the program builds tables that describe the functions and their references. Both the initialization and the construction phase search this information once for each node. Second, the algorithm inserts edges into the graph, which allows visiting of nodes in order to update their levels. Finally, it contributes to the building of function vector sets for all nodes by reference expansion. The contributions made by the Ryder's algorithm are proved to be of immense value in generating a call graph of a program, and its effectiveness is well-established in the field of program analysis.

The output of the Chaincode Analyzer produces a call graph in svg format and an AST in dot/pdf format. The source code of the analyzer and the obtained results can be found at GitHub[3]. Our Chaincode Analyzer, which is implemented in Node. js, has dependencies on the Graphviz library (Ellson et al., 2004), Google Closure Compiler (Bolin, 2010), and JDK. The Graphviz library is utilized for the visualization of the call graph, while the Google Closure Compiler is responsible for generating the AST. The output call graph is color-coded for convenience, with each color carrying a specific meaning. Functions highlighted in blue denote internal chaincode functions, while orange represents calls to external chaincode functions. Lastly, yellow is used to signify blockchain platform-dependent functions. Furthermore, the call graph of chaincodes of Hyperledger besu written in Solidity language is obtained using off-the-shelf tool surya (Surya, 2018).

7 Experimental evaluation

7.1 Performance analysis of the Chaincode Analyzer

We evaluated our tool on a variety of chaincodes written in the JavaScript language using a Windows-based desktop equipped with an Intel Core i7 CPU 3.00 GHz and 8 GB RAM. The benchmark chaincodes, shown in Table 2, include the codebase of the pub-sub

3 https://github.com/mdtauseefalam/JavaScriptChaincodeAnalyzer/

TABLE 2 Performance analysis of the Chaincode Analyzer.

Sl. No.	Chaincode	LoC	Chaincode functions details				Call graph generation	Memory usage
			# Internal functions	# External functions	# Blockchain-dependent functions	# Total functions	Time (in milliseconds)	(In MB)
1	broker.js	276	8	3	4	15	51	0.24
2	pubsub.js	194	4	0	3	7	39	0.11
3	topics.js	152	5	2	3	10	29	0.12
4	topics (F1.4).js	119	5	0	3	8	23	0.09
5	ERC20Token.js	503	16	0	4	20	100	0.77
6	ERC721Token.js	464	19	0	6	25	104	0.79
7	AssetTransfer.js	167	8	0	4	12	44	0.14
8	AssetTransferLedger.js	351	15	0	11	26	54	0.40
9	Abstore.js	138	5	0	4	9	28	0.10
10	FabCar.js	145	5	0	3	8	39	0.09

interoperability protocol, along with sample chaincodes imported from the Hyperledger project (Foundation, 2015). Observe that the chaincodes under rows (1)–(4) belong to the pub-sub interoperability protocol, while the chaincodes under rows (5)–(10) are few popular chaincodes which are available on the GitHub repository of the Hyperledger project. In columns 5 and 6, we report the performance evaluation results of our Chaincode Analyzer in terms of the time taken and the memory resources utilized to generate the call graph of the benchmark codes. Figures 9A, B depict the cost of call graph generation in terms of lines of code (LoC) v/s time and memory, respectively. This is to observe that, even though call graph generation time and memory usage depend on LoC, there are other factors, such as the number of functions present in the contract and the number of function calls, which may also influence them. For example, memory consumption in case of "ERC721Token.js" is more than that in "ERC20Token.js," although the LoC of the former one is less than that of the latter one. Furthermore, the number of functions in "AssetTransferLedger.js" (351 LoC) is more than that of "ERC20Token.js" (503 LoC) and "ERC721Token.js' (464 LoC). However, the execution time and memory consumption of "AssetTransferLedger.js" are significantly lower than those of "ERC20Token.js" and "ERC721Token.js". This is due to the number of function calls made in the code and the creation of edges in the corresponding call graph. Notably, the number of function calls in ERC721Token.js, ERC20Token.js, and AssetTransferLedger.js is 90, 73, and 49, respectively.

7.2 Properties verification results using UPPAAL-SMC

For verifying real-time properties, we simulate all these templates in UPPAAL-SMC where (1) the publisher (Fabric V2) is defined as Pub; (2) the broker (Fabric V2) has two chaincodes, connector and topics, which are defined as DispatcherConnector and DispatcherTopic, respectively; and (3) two subscriber networks (Hyperledger Fabric V1.4 and Hyperledger Besu) are defined as SubFabric and SubBesu, respectively. For the properties verified as follows, we assume that the pub-sub network architecture holds fairness property (i.e., the channels connecting publisher/subscriber blockchain to broker blockchain are reliable). We consider the following set of properties in TCTL for verifying the functional requirements and in MITL for verifying the non-functional requirements.

1. *A publisher/subscriber will be able to join the protocol eventually*. This property is expressed in TCTL as follows: (a) publisher joining the broker network: E◇ Pub.Registered imply DispatcherConnector.Registered and (b) subscriber joining the broker network: E◇ SubFabric.Registered imply DispatcherConnector.Registered
2. *The registered publisher can successfully create a new topic eventually*. This is expressed as follows: E◇ Pub.TopicCreated. The property in (2) is not enough to confirm whether the topic is created on the broker network. So, we consider the following property in (3).
3. *The topic created by the publisher is eventually created on the broker's network*. This is stated as follows: A◇ Pub.TopicCreated imply DispatcherTopic.TopicCreated
4. *Messages published to a topic by the publisher are always received by the subscribers eventually*. It is represented as follows: A◇ ((Pub.Published and SubFabric.Subscribed) imply SubFabric.Notified)
5. *Creation of duplicate topics by the publisher will eventually lead to the creation of duplicate topics by the broker*. This is stated in TCTL as follows: A◇ (Pub.Registered and Pub.TopicCreated and `(Pub.Ptop_id == 0)` and `(DispatcherTopic.Btop_id == 0)` imply `(DispatcherTopic.TopicCreated and (DispatcherTopic.Btop_id = = 0)))`

Observe that the properties (1) and (2) represent reachability properties, whereas the properties (3), (4), and (5) represent liveness

TABLE 3 Probability estimated for message interoperability between the publisher and subscriber blockchains.

Sl.no.	Property	Throughput (TPS)			Probability estimated	Time in seconds
		Publisher	Broker	Subscriber		
1	Pr (<>[10,100] (Sub.Subscribed And Pub.Published) Imply Sub.Notified)	20,000	20,000	300	[0.92019,1]	0.019
2		10,000	20,000	20,000	[0.898509,0.998509]	0.018
3		20,000	20,000	20,000	[0.910705,1]	0.017
4		2,000	10,000	3,000	**[0.92561,1]**	0.016
5		2,500	2,000	3,000	[0.160027,0.260027]	0.015
6		10,000	2,000	20,000	**[0.153252,0.253252]**	0.015

The bold values Represent highlight the cases where the pub-sub interoperability protocol performs the best (i.e. [0.92561,1]) and worst (i.e. [0.153252, 0.253252]) within a given time units. The values tell the probability of the same.

properties. The verification results by UPPAAL-SMC demonstrate that the properties in (1)–(5) are satisfied by the aforementioned model. These results provide insightful information to the end-users to decide the adaptability of the protocol in a particular application depending on its interoperability needs. This is to observe that, in order to avoid the state explosion issue, we consider five topics in our model, and we verify these properties against all these five topics. However, if one wishes to check a property only for a particular topic, the corresponding topic id should be used in the TCTL formula[4]. For example, property (3) can be rewritten as follows: $A\Diamond$ `((Pub.TopicCreated and (Ptopic[0]==true)) imply (DispatcherTopic.TopicCreated and` (Btopic [0] = = true)))

Let us now estimate the probabilities of properties within a given time span using the following MITL temporal logic:

6. *What is the probability of the topic being created on the publisher network within 45 time units?* This is represented as follows: Pr ($\Diamond$ $[0, 45]$ `Pub.TopicCreated`). This property is satisfied with the probability of [0.0990515, 0.199051] with 95% level of significance in 738 runs.
7. *What is the probability of a topic being created on the broker's network within 100 time units when it has already been created by the publisher?* This is expressed as follows: Pr [< = 100] ($\Diamond$ `Pub.TopicCreated imply DispatcherTopic.TopicCreated`). This property gives probabilistic estimation of [0.901855, 1] with 95% level of significance in 29 runs.
8. *What is the probability of the subscribers to receive the messages published by the publisher in the time interval of 10 to 100 time-units?* This is stated as follows: Pr ($\Diamond$ $[10, 100]$ `(SubBesu.Subscribed & Pub.Published) imply SubBesu.Notified`). This gives an estimate of the probability in the range of [0.92019, 1] with 95% level of significance in 738 runs. This reveals the fact that the protocol may fail to transmit messages between a publisher and a subscriber with 8% probability when the throughput of the broker blockchain is relatively higher than that of both the publisher and subscriber, within a given time range of 10–100 μs (microseconds). Such information is generally valuable for the end-users as they make decisions regarding the adoption of the pub-sub protocol in hard real-time decentralized applications. Table 3 depicts the verification result of this property by varying the throughput of the publisher, broker, and subscriber networks. This is to observe that the protocol gives a best result when the throughput of the broker blockchain is comparatively higher than that of the publisher and the subscriber.
9. *Is the probability of publishing (by the publisher) a message to a topic which gets reflected on the broker network equal to the probability of publishing a message (by the broker) to the associated subscribers within 100 time units?* which is expressed as follows: Pr [< = 100] ($\Diamond$ `(SubFabric.Subscribed and Pub.Published) imply DispatcherTopic.Published)` > = Pr [< = 100] ($\Diamond$ `DispatcherTopic.Published imply SubFabric.Notified`). This property aims at verifying the possibility of message loss within a given time frame. The verification result of this property yields the result shown in Figure 10, where 0.5 means both the probability is the same. Therefore, we conclude that there is no message loss in the protocol.

8 Threats to validity

Let us first discuss the threats to external validity in our study, which relate to the generalization of our findings. One major concern is that certain assumptions we have made may not hold true in a real-world scenario. Given that Hyperledger blockchain networks are mature and established, we have assumed that they are safe and secure, thereby abstracting the block generation process and consensus algorithm in our proposed model. Additionally, to mitigate the state space explosion issue, we have limited our

4 The variable used in the TCTL formula must be globally declared

experiments to a fixed number of parameters, for instance, the number of topics, which may vary for different application scenarios.

Moreover, we have made a number of assumptions regarding the absence of issues stemming from network hardware components, which may lead to communication delay. Although we consider a uniform network delay in our modeling process, this may not be the reality. In addition, our assumption on the fairness of the communication channels hinders us to verify the properties in presence of network failure.

Considering the aforementioned assumptions, we followed a systematic approach for modeling the protocol to ensure that the actual code and the model exhibit the same behavior. In particular, we build an abstract STA model from the source code of the protocol by extracting relevant information guided by our properties of interest. It is important to note that this abstract model may not be sufficient to verify any new property due to the over-approximation behavior. In such cases, refinement may be necessary to include property-specific information in the model. Lastly, it is essential to highlight that our Chaincode Analyzer currently faces limitations in handling recursion-based functions smoothly due to the implemented Ryder's algorithmic constraints.

We will now delve into the threat to internal validity, which refers to the potential experimental bias and errors that may arise due to our implementation. We acknowledge that the call graph construction algorithm (Ryder, 1979) used by our tool was originally designed for FORTRAN programs, but we have ensured that it works seamlessly for JavaScript chaincodes as well, considering their similar function construct. Nevertheless, we must also recognize that converting call graphs to STA may require human intervention, which may result in model biases.

9 Conclusion

In this paper, we have formally modeled the pub-sub-based blockchain interoperability protocol in the UPPAAL-SMC model checker and verified its functional real-time properties. Along with this, we have estimated the probabilities of various properties in a given time limit and highlighted the cases where the pub-sub interoperability protocol performs the best and worst. The verification results revealed that no message loss takes place during inter-blockchain communication and provides an insight on choosing appropriate blockchain networks with suitable throughput as publishers, brokers, or subscribers, ensuring the highest utility of the protocol. Observe that, even though the model satisfies some of our properties of interest, there are certain cases where probability of failure is observed. For example, our verification reveals the fact that the protocol may fail to transmit messages between a publisher and a subscriber with 8% probability when the throughput of the broker blockchain is relatively higher than that of both the publisher and subscriber, within a given time range of 10–100 μs (microseconds). Such information is generally valuable for the end-users as they make decisions regarding the adoption of the pub-sub protocol in hard real-time decentralized applications. Furthermore, as the access control mechanism is yet to be addressed by the protocol and is part of their future work, the verification of the security aspects of the protocol would be an interesting future scope.

Data availability statement

The original contributions presented in the study are included in the article/Supplementary Material; further inquiries can be directed to the corresponding authors.

Author contributions

Conceptualization: MA, RH, and AM; methodology: MA; formal analysis: MA and RH; writing—original manuscript: MA; writing—review and editing: MA, RH, and AM; supervision: RH and AM. All authors contributed to the article and approved the submitted version.

Acknowledgments

We graciously acknowledge the support of the Prime Minister Research Fellowship (PMRF) Award and the SERB Core Research Grant (Grant Number: CRG/2022/005794) by the Government of India for carrying out this research.

Conflict of interest

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's note

References

Abdellatif, T., and Brousmiche, K.-L. (2018). "Formal verification of smart contracts based on users and blockchain behaviors models," in *2018 9th IFIP international conference on new Technologies, mobility and security (NTMS)* (IEEE), 1–5. doi:10.1109/NTMS.2018.8328737

Acorn (2014). Acorn. Available at: https://github.com/acornjs/acorn.

Afzaal, H., Imran, M., Janjua, M. U., and Gochhayat, S. P. (2022). Formal modeling and verification of a blockchain-based crowdsourcing consensus protocol. *IEEE Access* 10, 8163–8183. doi:10.1109/access.2022.3141982

Aggarwal, S., Chaudhary, R., Aujla, G. S., Kumar, N., Choo, K.-K. R., and Zomaya, A. Y. (2019). Blockchain for smart communities: applications,

challenges and opportunities. *J. Netw. Comput. Appl.* 144, 13–48. doi:10.1016/j.jnca.2019.06.018

Alqahtani, S., He, X., Gamble, R., and Mauricio, P. (2020). "Formal verification of functional requirements for smart contract compositions in supply chain management systems," in *Proc. Of the 53rd Hawaii international conference on system sciences*, 5278–5287.

Andrychowicz, M., Dziembowski, S., Malinowski, D., and Mazurek, Ł. (2014). "Modeling bitcoin contracts by timed automata," in *International conference on formal modeling and analysis of timed systems* (Springer), 7–22.

Atzei, N., Bartoletti, M., Lande, S., and Zunino, R. (2018). "A formal model of bitcoin transactions," in *Financial cryptography and data security: 22nd international conference, FC 2018, nieuwpoort, curaçao, february 26–march 2, 2018, revised selected papers 22* (Springer), 541–560.

Bai, X., Cheng, Z., Duan, Z., and Hu, K. (2018). "Formal modeling and verification of smart contracts," in *Proceedings of the 2018 7th international conference on software and computer applications*, 322–326.

Bartoletti, M., and Zunino, R. (2019). Formal models of bitcoin contracts: A survey. *Front. Blockchain* 2, 8. doi:10.3389/fbloc.2019.00008

Behrmann, G., David, A., and Larsen, K. G. (2004). A tutorial on uppaal. *Formal methods Des. real-time Syst.* 3185, 200–236. doi:10.1007/978-3-540-30080-9_7

Belchior, R., Vasconcelos, A., Guerreiro, S., and Correia, M. (2021). A survey on blockchain interoperability: past, present, and future trends. *ACM Comput. Surv. (CSUR)* 54, 1–41. doi:10.1145/3471140

Bengtsson, J., and Yi, W. (2004). *Timed automata: Semantics, algorithms and tools*. Berlin, Heidelberg: Springer Berlin Heidelberg, 87–124.

Bertrand, N., Gramoli, V., Konnov, I., Lazic, M., Tholoniat, P., and Widder, J. (2022). "Brief announcement: holistic verification of blockchain consensus," in *Proceedings of the 2022 ACM symposium on principles of distributed computing*, 424–426.

Bolin, M. (2010). *Closure: The definitive guide: Google tools to add power to your javascript*. O'Reilly Media, Inc.

Cassandras, C. G., and Lafortune, S. (1999). "Stochastic timed automata," in *Introduction to discrete event systems* (Springer), 317–365.

Chaudhary, K. C., Chand, V., and Fehnker, A. (2020). "Double-spending analysis of bitcoin,", in *Pacific asia conference on information systems proceedings (association for information systems)*.

Chaudhary, K., Fehnker, A., Van De Pol, J., and Stoelinga, M. (2015). *Modeling and verification of the bitcoin protocol. arXiv preprint arXiv:1511.04173*.

Clarke, E. M., Grumberg, O., and Long, D. E. (1994). Model checking and abstraction. *ACM Trans. Program. Lang. Syst.* 16, 1512–1542. doi:10.1145/186025.186051

David, A., Larsen, K. G., Legay, A., Mikučionis, M., and Poulsen, D. B. (2015). Uppaal smc tutorial. *Int. J. Softw. Tools Technol. Transf.* 17, 397–415. doi:10.1007/s10009-014-0361-y

DiGiacomo-Castillo, M., Liang, Y., Pal, A., and Mitchell, J. C. (2020). "Model checking bitcoin and other proof-of-work consensus protocols," in *2020 IEEE international conference on blockchain (blockchain)* (IEEE), 351–358.

Dreyer, J., Fischer, M., and Tönjes, R. (2020). "Performance analysis of hyperledger fabric 2.0 blockchain platform," in *Proceedings of the workshop on cloud continuum services for smart IoT systems*, 32–38.

Eijkel, D., and Fehnker, A. (2019). "A distributed blockchain model of selfish mining," in *International symposium on formal methods* (Springer), 350–361.

Ellson, J., Gansner, E. R., Koutsofios, E., North, S. C., and Woodhull, G. (2004). "Graphviz and dynagraph—Static and dynamic graph drawing tools," in *Graph drawing software*, 127–148. doi:10.1007/978-3-642-18638-7_6

Esprima (2015). Esprima. Available at: https://github.com/jquery/esprima.

Fehnker, A., and Chaudhary, K. (2018). "Twenty percent and a few days–optimising a bitcoin majority attack," in *NASA formal methods symposium* (Springer), 157–163.

Foundation, T. L. (2015). Hyperledger foundation. Available at: https://www.hyperledger.org/.

Ghaemi, S., Rouhani, S., Belchior, R., Cruz, R. S., Khazaei, H., and Musilek, P. (2021). *A pub-sub architecture to promote blockchain interoperability. arXiv preprint arXiv: 2101.12331*.

Gu, X., Cao, W., Zhu, Y., Song, X., Huang, Y., and Ma, X. (2022). *Compositional model checking of consensus protocols specified in tla+ via interaction-preserving abstraction. arXiv preprint arXiv:2202.11385*.

Hewa, T., Ylianttila, M., and Liyanage, M. (2021). Survey on blockchain based smart contracts: applications, opportunities and challenges. *J. Netw. Comput. Appl.* 177, 102857. doi:10.1016/j.jnca.2020.102857

Khan, A. G., Zahid, A. H., Hussain, M., Farooq, M., Riaz, U., and Alam, T. M. (2019). "A journey of web and blockchain towards the industry 4.0: an overview," in *2019 international conference on innovative computing (ICIC)*, 1–7. doi:10.1109/ICIC48496.2019.8966700

Liu, Y., Zhou, Z., Yang, Y., and Ma, Y. (2022). Verifying the smart contracts of the port supply chain system based on probabilistic model checking. *Systems* 10, 19. doi:10.3390/systems10010019

Mavridou, A., Laszka, A., Stachtiari, E., and Dubey, A. (2019). "Verisolid: correct-by-design smart contracts for ethereum," in *International conference on financial cryptography and data security* (Springer), 446–465.

Nakamoto, S. (2008). Re: bitcoin p2p e-cash paper. *Cryptogr. Mail. List*.

Nam, W., and Kil, H. (2022). Formal verification of blockchain smart contracts via atl model checking. *IEEE Access* 10, 8151–8162. doi:10.1109/access.2022.3143145

Nehai, Z., Piriou, P.-Y., and Daumas, F. (2018). "Model-checking of smart contracts," in *2018 IEEE international conference on iThings & GreenCom & CPSCom & SmartData* (IEEE), 980–987.

Osterland, T., and Rose, T. (2020). Model checking smart contracts for ethereum. *Pervasive Mob. Comput.* 63, 101129. doi:10.1016/j.pmcj.2020.101129

Park, W. S., Lee, H., and Choi, J.-Y. (2022). "Formal modeling of smart contract-based trading system," in *2022 24th international conference on advanced communication technology (ICACT)* (IEEE), 48–52.

Pnueli, A. (1977). "The temporal logic of programs," in *18th annual symposium on foundations of computer science (sfcs 1977)* (IEEE), 46–57.

Ryder, B. (1979). "Constructing the call graph of a program," in *IEEE transactions on software engineering SE-5*, 216–226. doi:10.1109/TSE.1979.234183

Surya, the sun god: A solidity inspector (2018). Surya, the sun god: A solidity inspector. Available at: https://github.com/ConsenSys/surya.

The callgraphjs tool (2014). The callgraphjs tool. Available at: https://github.com/asgerf/callgraphjs.dart.

The code2flow tool (2021). The code2flow tool. Available at: https://github.com/scottrogowski/code2flow.

The javascript explorer callgraph tool (2018). The javascript explorer callgraph tool. Available at: https://github.com/shrivastava-apurva/Javascript-Explorer—Callgraph.

Tolmach, P., Li, Y., Lin, S.-W., Liu, Y., and Li, Z. (2021). A survey of smart contract formal specification and verification. *ACM Comput. Surv. (CSUR)* 54, 1–38. doi:10.1145/3464421

Zhang, Q., Lu, Y., and Sun, M. (2020). "Modeling and verification of the nervos ckb block synchronization protocol in uppaal," in *International conference on blockchain and trustworthy systems* (Springer), 3–17.