Exploring Hyperledger Sawtooth: Model Checking Proof-of-Elapsed Time Algorithm and Testing Methods for Enterprise Blockchain Applications

Sergey Staroletov
Polzunov Altai State Technical University
Barnaul, Russian Federation
serg_soft@mail.ru

***Abstract*—Blockchain 3.0 or distributed ledger applications are decentralized, autonomous organizational units governed by their laws. The distributed ledger here means that all records (for example, application data, function call results) are stored in a distributed way among nodes (or peers) of a private network. Cryptocurrencies are a specific application of distributed ledger technology focused on digital currency transactions, while distributed ledger applications have broader use cases beyond finance and other enterprise activities, using blockchains to enhance trust and transparency across various industries. This work aims to illustrate Hyperledger Sawtooth, which is now becoming an integral part of the Splinter platform, as a key example of a Blockchain 3.0 solution.**

We propose methodologies for modeling and verifying its consensus protocol and discuss the application testing for this platform using a containerization approach. We discuss key insights related to enterprise applications, the Hyperledger consortium, and the various components of the Hyperledger project. We provide a brief overview of Hyperledger architecture, focusing on the PoET (Proof-of-Elapsed Time) consensus protocol and the hardware methods employed to ensure its reliability. Additionally, we address formal verification techniques, particularly Model Checking, to verify the correctness of a simplified protocol model. Finally, we introduce our industrial solution for testing Sawtooth applications using Docker containerization.

I. Introduction

A blockchain application is such kind of modern software, whose instance stores its own blockchain state, synchronized to chains of blocks among other network participants (instances of the same application), and for the synchronization, a consensus algorithm is used. Some analysis of such applications in presented in [1]. These applications can be categorized into three major groups: 1.0, 2.0, and 3.0 (please do not confuse with the web3 concept in the Ethereum project discussed in [2]):

1) *Blockchain 1.0* refers to cryptocurrencies or digital currencies that serve as alternatives to traditional fiat currencies (e.g., EUR or USD), with Bitcoin being the first and most well-known example.
2) *Blockchain 2.0* encompasses models based on “smart contracts” (for example, see the review [3]), which are code lines written in specialized programming languages that automatically carry out predetermined processes without the need for an intermediary, such as a bank.
3) *Blockchain 3.0* includes distributed ledger applications that function as decentralized, self-governing organizational units operated by their own rules (see some definitions in [4]). These applications can be utilized in business contexts and even within public governance frameworks, offering reliable data storage, access, and verifiable services like voting and document signing.

The distributed ledger here means that all records (for example, application data, function call results to modify current state) are stored in a distributed way among nodes (or peers) of a private network.

The goal of the work it to discuss Hyperledger Sawtooth as an example of the Blockchain 3.0 platform and propose methods to model and verify their consensus protocol and provide testing methods for platform applications using the containerization approach.

In the Background section, we highlight notable information about enterprise applications, Hyperledger consortium and components of the Hyperledger project, then we describe Hyperledger architecture, the consensus protocol PoET (Proof-of-Elapsed Time) and hardware methods to ensure the trustworthy of the protocol proposed by Intel corporation. We refer to formal verification methods, in particular, to Model Checking and in the appropriate section, we verify the correctness of a protocol model. We develop a formal model for the PoET which can be used as a basis for further vulnerability checks of the protocol and for the educational purposes.

In the section ”Testing Methods of Enterprise Blockchain Applications”, we propose our industrial solution to test Sawtooth applications using the containerization with Docker.

II. Related works

In Blockchain 3.0 world, currently known a number of implemented proof-of-concept solutions using Hyperledger Fabric platform including e-voting system [5], a cost-effective solution for healthcare [6], a solution for power grids in a secure, controlled, monitored, and efficient manner [7]. In the book [8] an introduction to programming with the Fabric platform is given.

Regarding blockchain applications testing, in the work [9] we focused on the containerization when testing Blockchain 1.0 applications as virtual currency gateways, some existing

testing methods and frameworks for Bitcoin and Ethereum were reviewed and results of tests were discussed.

Moving to consensus protocols, a review of these protocols was conducted in the paper [10]. In [11], the Proof-of-Elapsed Time protocol (PoET) was discussed and modeled using probability distributions for the purposes of security analysis. As PoET is based on Intel SGX extensions, we also note the paper [12]. A possible attack to this platform is discussed in [13].

III. Background

A. Enterprise-level blockchain applications

As we have previously noted, blockchain technology has evolved beyond its initial association with cryptocurrencies. Today, it offers a diverse range of applications that can significantly enhance various aspects of human life. While cryptocurrencies demonstrated the practical potential of blockchain, many organizations have begun adopting its functionalities in controlled environments, leading to innovative solutions in corporate settings. We refer here a special terminology to categorize the current blockchain types (discussed in [14]):

- Permissionless blockchains allow free access and interaction without the need for special permissions. These blockchains maintain their security through consensus algorithms, which are essential to their operation. Central to these algorithms is the concept of rewarding miners (nodes that support the network by validating transactions and signing blocks according to specific rules and functions). Popular examples of permissionless blockchains include well-known cryptocurrencies like Bitcoin and Ethereum.
- Permitted blockchains, also known as permissioned blockchains or consortium blockchains [15], are those whose infrastructure is managed by their creators. In these systems, the consensus mechanism that relies on miners for security is less relevant, as they are typically non-public or read-only. If a token (unit of account) exists within this type of blockchain, it is solely for technical purposes; it cannot be withdrawn or used outside the network, rendering it devoid of real value. Despite these limitations, permitted blockchains share key characteristics with their permissionless counterparts, such as robust security, immutability of transaction history, and decentralization. Additionally, they are also capable of executing smart contracts.

The most popular tools now for creating permitted-blockchain systems are introduced by the Hyperledger consortium.

B. Hyperledger consortium

Hyperledger is a joint open-source project created to promote blockchain technologies by the implementation of common functions necessary for an open cross-industry standard of distributed ledgers. This is an international project that brings together leading companies in the field of finance, banking, Internet of Things, logistics, manufacturing and technology. The Hyperledger project operates with the support of Linux Foundation [16].

The Hyperledger project began its existence at the end of 2015 within the framework of the Linux Foundation (known by ensuring the development of not only the famous operating system but also, for example, the Node.js platform). In 2016, twenty companies and organizations joined the project, including IBM, which transferred the source code of OpenBlockchain (later renamed to Fabric), and then Intel Corporation came with the code of Sawtooth [16].

In general, Hyperledger aims to develop open distributed ledger technologies that enable companies to build stable industry applications, platforms, and hardware systems designed to perform specific business operations. The Hyperledger initiative originally featured a consortium of approximately 100 partner organizations, representing a diverse of industries. Corporate members included aerospace leaders like Airbus, automotive giants such as Daimler, and technology companies including Fujitsu, Huawei, Nokia, and Samsung. The financial sector was represented with prominent institutions like J.P. Morgan and Wells Fargo. The strategy to unite a set of industry players under a shared framework can accelerate the technological development of blockchain as well establish best practices and standards that could benefit the wider adoption of the technology in various sectors.

Currently, Hyperledger offers some major services [17]:

1) Sawtooth[1], initially developed by Intel, is designed for the creation, implementation, and management of public digital ledgers. It includes a unique consensus mechanism known as “Proof-of-Elapsed Time” (PoET), which is the focus of our paper and aims to reduce resource consumption across a large decentralized network of validators. Sawtooth facilitates the development of both permissioned and permissionless blockchains. Currently, the Sawtooth project has reached the end of its development and is planned to be integrated into the Splinter platform (a privacy-focused platform for distributed applications).
2) Fabric[2], initially developed by IBM, serves as a framework for crafting private, permissioned document-oriented blockchains. Developers working with Fabric have the flexibility to select from various consensus algorithms, blockchain data formats, and user authentication methods. Additionally, it supports smart contracts written in the Go programming language.
3) Iroha[3] framework enables to support various applications, from private blockchains utilized by central banks to public blockchains designed for NFT and cryptocurrency trading. Iroha was developed with the capability for cross-blockchain communication in mind. Its event-driven architecture aids in streamlining operations.
4) Indy[4], developed by the Sovrin Foundation, is a software

[1] https://sawtooth.splinter.dev/docs/1.2/
[2] https://hyperledger-fabric.readthedocs.io/en/release-2.5/
[3] https://iroha.tech
[4] https://hyperledger-indy.readthedocs.io/projects/indy/en/latest/

development kit (SDK) designed for managing "digital passports" (Self-Sovereign Identity) and facilitating their integration with public blockchains.

The Hyperledger components currently exhibit a degree of fragmentation, primarily due to their transfer from various organizations to the consortium at different points in time. This lack of cohesion can pose challenges for integration and collaboration within the ecosystem. The fragmentation of Hyperledger components raises important considerations for organizations looking to adopt blockchain technology. In this paper, we choose to study the Hyperledger Sawtooth framework, due to its enterprise-grade blockchain nature designed for scalability and flexibility, support a range of permissions and consensus mechanisms appropriate for various business use cases including Proof of Elapsed Time (PoET), which is energy-efficient and designed for permissioned networks.

C. Hyperledger Sawtooth architecture

Sawtooth is a framework for the development of enterprise-level distributed ledger systems. Proposed by Intel, declared to be focused on security, scalability and modularity [18]. The Sawtooth architecture consists of the following main components [18] (see also [5]) :

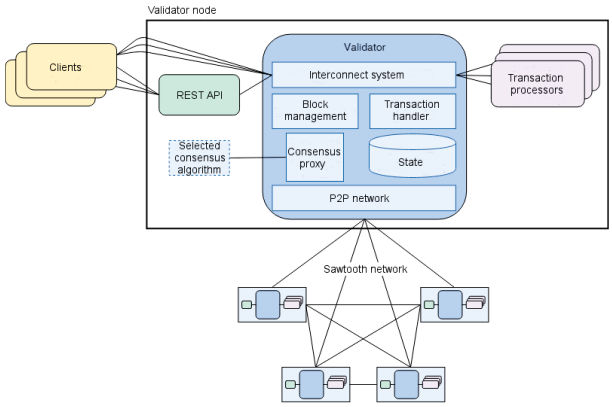

Fig. 1. Architecture of the Hyperledger Sawtooth network organization

1) *Peer-to-Peer network* for sending messages and transactions between nodes.
2) *Distributed ledger* contains ordered transactions.
3) *State machine/smart contract logic layer* to handle transaction content.
4) *Distributed storage* based on Merkle trees.
5) *Consensus algorithm* to determine the order of transactions and the resulting state.
6) *Transaction Processor* to manage the business logic of transactions by validating and applying them to the blockchain's state. Sawtooth comes with predefined transaction families and also supports the creation of new ones.
7) *Validator* is a node within the network responsible for validating transactions and proposing new blocks. It ensures that transactions are processed accurately and that blocks comply with the consensus criteria.

In Fig. 1 (obtained from the documentation), we show how the components are connected and interact with each other. In Fig. 2, we depict our reconstruction of the internal architecture of the project by studying its source code.

D. PoET

Proof-of-Elapsed Time (PoET) is a specific consensus protocol, which can be used in the Hyperledger Sawtooth network to choose a winner on the next generation unit. Intel developed the PoET algorithm to offer a fair competition mechanism for nodes seeking to generate new blocks in a blockchain. This approach prioritizes efficiency and simplicity, and it achieves a significantly lower energy consumption compared to conventional consensus methods, such as Proof of Work (PoW). The operation of the PoET algorithm is as follows: each participating node in the network should wait for a random period of time, and the first one who wakes up after the waiting gets the right to generate a new block. That is, the process can be schematically represented as (with some degree of abstraction):

1) Each node in the blockchain network generates a random timeout and goes into sleep mode for a specified period.
2) The one that wakes up first, i.e. the node with the shortest wait time, becomes eligible to create a new block in the blockchain, broadcasting the necessary information across the peer-to-peer network.
3) The block is verified by other participants.
4) The same process is repeated to find the next block.

The consensus mechanism of the PoET network must meet two important conditions: (1) the waiting time should really be chosen by the participants randomly, and not based on some short period, which increases the chances of winning; and (2) the winner should really wait until the end of the appointed time.

Thus, PoET offers a high-tech approach to solve the computational problem of "random leader election" (see some definitions [19] and a survey [20] on it). To maintain fairness and prevent manipulation of wait times, the PoET mechanism can utilize a trusted execution environment implemented in Intel Software Guard Extensions (SGX). It ensures that the random wait time is generated and processed securely. Once the node awakens and proposes a block, it provides proof of its wait time, which can be verified by other nodes within the network.

E. Intel SGX

The SGX (Software Guard Extensions) technology provides a means to execute trusted code within secure enclaves, ensuring a high level of isolation and protection against unauthorized access. This technology comprises several components: specialized hardware instructions, a kernel module, user-space code that interacts with the kernel module, and developer tools designed for creating, declaring, and signing trusted components of the code, as well as for verifying their integrity.

[5]https://www.geeksforgeeks.org/hyperledger-sawtooth-in-blockchain/

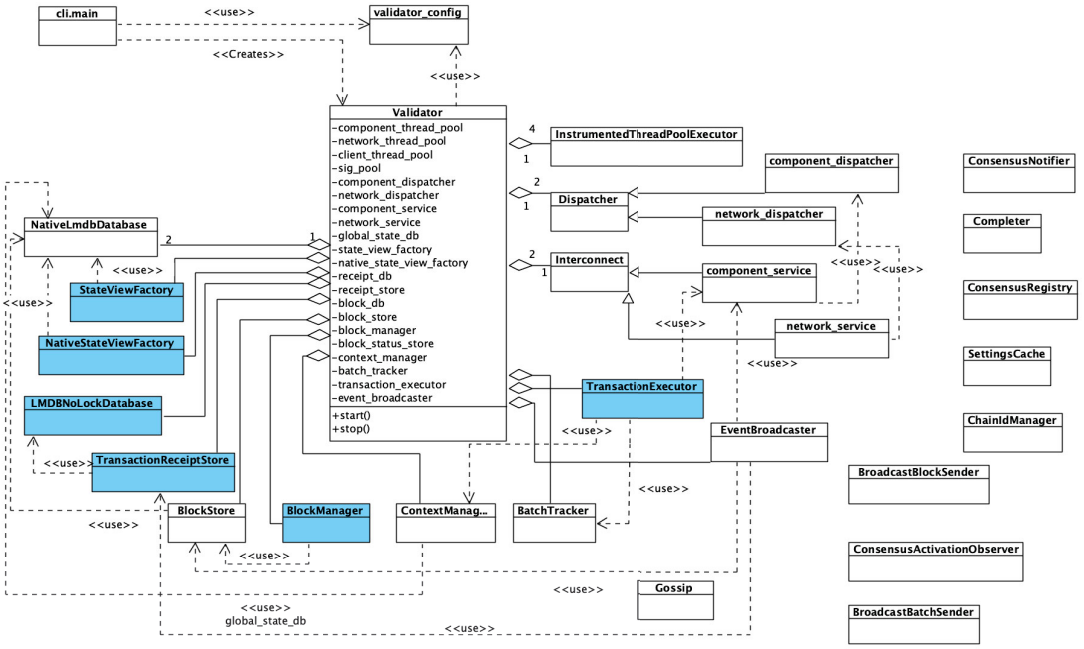

Fig. 2. Internal Architecture of the Hyperledger Sawtooth project

With SGX, developers can design solutions that rely on isolated and immutable code segments. This is useful for applications requiring a trustworthy execution environment, such as those utilizing the PoET protocol for trusted waiting mechanisms.

In Fig. 3, we show a fragment of PoET enclave definition file (.edl). Please follow the documentation to get the definitions for used functions like *CreateWaitTimer()*[6]). The Hyperledger Sawtooth SGX implementation is available on a Github repository [21]. The developer documentation is available from Intel [22].

While SGX aims to create a secure enclave, numerous vulnerabilities have emerged over time, including Foreshadow [23]. The effectiveness of SGX is largely contingent upon the security of the host platform, requiring the processor to be free from vulnerabilities that could be exploited either directly or indirectly. Consequently, new security advancements like Intel's Trust Domain Extensions (TDX) and AMD's Secure Encrypted Virtualization (SEV) offer hardware-based isolation and improved capabilities for virtualization, making SGX from the 11th generation of Intel Core processors deprecated. An alternative to Intel SGX from ARM is ARM TrustZone.

Thus, in addition to the initial PoET specification, in the face of vulnerabilities, a special statistical test can be used in the protocol to determine whether the waiting time of the leader really follows the specified distribution [11].

F. Formal verification with SPIN tool

Formal verification allows verifying engineers to prove the correctness of a model on all possible states with respect to given requirements. The Model Checking method deals with models in the form of programs that are converted to a special kind of finite state machines, and the requirements expressed in terms of linear-time temporal logic (LTL). Teaching this can be combined with software testing in modern university courses [24]. The logic was first introduced in [25]. Since then, many different extensions have been made (including domain-specific ones [26]), but the verifier we use only supports pure LTL.

SPIN (stands for Simple Promela INterpreter) is a formal verification tool for models written in Promela (Protocol MEta-LAnguage) with respect to given LTL requirements (formulas constructed using key variables of the model). To deal with our models, we may rely upon the following language features [27]:

- it is an actor-based (process-oriented) language;
- it is primarily designed to describe protocols interoperations;
- it has C-styled syntax;
- it allows non-deterministic transitions.

We use the SPIN tool and the Promela language to create and check a model of the PoET protocol behavior. We have already discussed that the protocol is a part of the leader-selection protocol family (such as one included as a case study in the SPIN repository [28]) – in PoET, a node is elected as a leader and then that node generates a new block in the blockchain.

Although SPIN and Promela are valuable tools for formal verification, their application to modeling complex real-world protocols like PoET necessitates careful consideration of their limitations in terms of complexity, state space management,

[6]https://sawtooth.splinter.dev/docs/core/1.2/architecture/poet.html

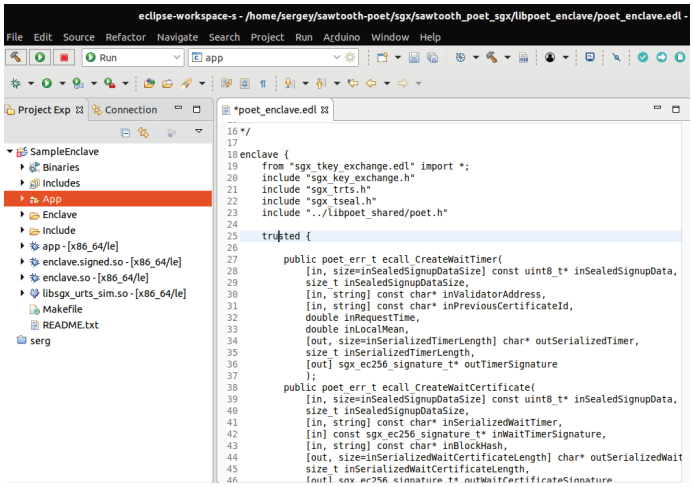

Fig. 3. Enclave definition in the Eclipse environment with Intel SGX tools installed

timing, expressiveness, and usability. In our work, we are balancing these factors to successfully leveraging the benefits of formal methods in practical situations.

G. DevOps methodics

DevOps is a combination of cultural principles, approaches and tools that allow software companies to create applications and services at high speed. With DevOps, product development and optimization are executed faster than using traditional software development and infrastructure management processes.

In the DevOps model, the lines between the development and operations teams become increasingly blurred. Often, these two groups merge into a single team, where engineers are responsible for the entire application lifecycle, from development and testing to deployment and maintenance. To improve efficiency, these teams leverage specialized methods, including dedicated software and automated scripts, to streamline processes that were once carried out manually and slowly. With these tools at their disposal, technicians can independently resolve issues that previously required assistance from other teams, such as deploying code or initializing infrastructure. This autonomy significantly accelerates overall workflow and productivity.

DevOps techniques and processes are the following [29]:

- *Continuous Integration* is a software development practice in which developers consistently merge their code changes into a central repository, followed by automated building and testing processes.
- *Continuous Delivery* builds upon continuous integration by guaranteeing that all code changes, following the build stage, are deployed to either a testing or production environment. When implemented effectively, continuous delivery ensures that developers always have a ready-to-deploy version of the software that has successfully passed standardized testing procedures [30].
- *Microservice Architecture* is a design approach that structures an application as a collection of small independent services. Each service operates within its own process and communicates with other services through a well-defined API, typically utilizing an HTTP-based interface. These microservices are tailored to meet specific business needs, with each one dedicated to performing a particular function.
- *Infrastructure as Code* is a modern practice that involves managing and provisioning infrastructure through code and software development techniques, such as version control and continuous integration. This approach enables engineers to interact with the infrastructure using code-based tools, just as they do with application code.

In this work, we apply these principles to develop, deploy and test Hyperledger Sawtooth applications using containerized Docker environment.

IV. Verification of PoET protocol using the Model Checking approach

We consider a very simplified implementation of the protocol that does not lose key properties of it. Fig. 4 shows the sequence of interaction that we model. To demonstrate the feasibility of the approach, we did the following: (1) implemented the PoET interaction sequence in the Erlang actor-based language[7]; (2) rewrote this code for channel interaction, which the Promela language allows. For ease of implementation in Promela, all P2P interactions are reduced to creating one network process (in reality, it is needed to send messages to all network nodes and, accordingly, respond to them).

So, the PoET process waits for the network to enter the initialization state of a new round, then it asks for a random number and waits for the specified time. After that, the current state of the network is determined, if there is no new block at the moment, the process considers ourselves the leader and checks the status of all other nodes to count the number of all nodes in leader and non-leader state (here it should be

[7]https://github.com/SergeyStaroletov/PromelaSamples/blob/master/PoET.erl

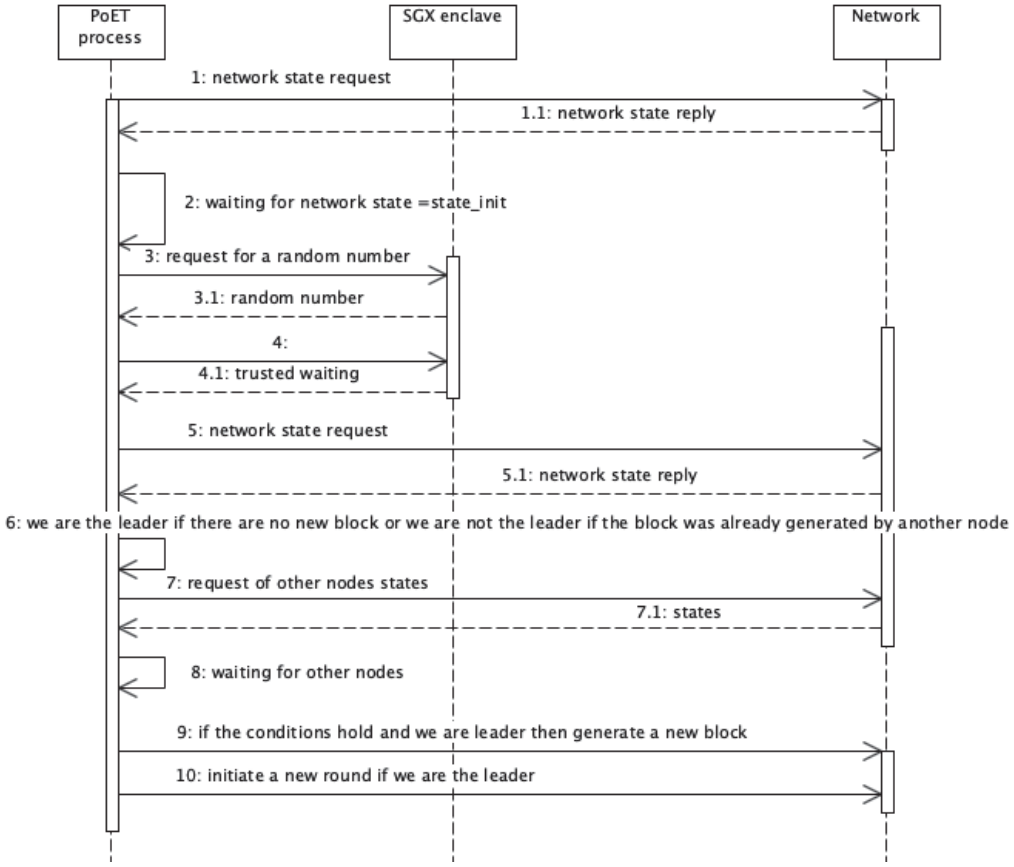

Fig. 4. Sequence diagram for block generation in PoET

noted that while the process was receiving a message, someone another could also become a leader), so the process needs to make sure that other nodes are in the expected state and then it can create a new block. If there are several leaders, the round is canceled and the process repeats. We understood the necessity of reelection during implementation and simulation of the model.

In this section, we include our Promela implementation of PoET model: it is not so complicated, easy-understandable and may show the Promela language features to an interested reader because one of the purposes of the paper was the popularization of Model Checking approach in the industrial sphere. The full code of the model is available on Github [31]. In the implementation, we use global variables for the network state instead of broadcasting this through the processes in messages.

To begin, we follow the SPIN guidelines [32] to random number generation and implement a non-deterministic process that simulates the operation of Sawtooth SGX enclave and generates random numbers from 0 to 32767 after requesting it from its channel:

```
active proctype gen() {
byte buf;
short nr;
do
  :: {
  SGX ? buf;
  do
        :: (nr < 32768) -> nr++;
        :: (nr > 0) -> nr--;
        :: break
  od;
  SGX ! nr;
  }
od
}
```

Next, we describe the global variables of the model, namely, we have five processes, the logic of which is modeled with all possible switches (interleaving):

```
#define P 5 /* number of processes in the
    model */

chan SGX = [0] of {short}; /* channel for
    random number generation */

mtype = {STATE_INIT, STATE_GEN,
   STATE_WAITING, STATE_MINE_BLOCK,
   STATE_NO_MINE_BLOCK}; /* type for
   states */
```

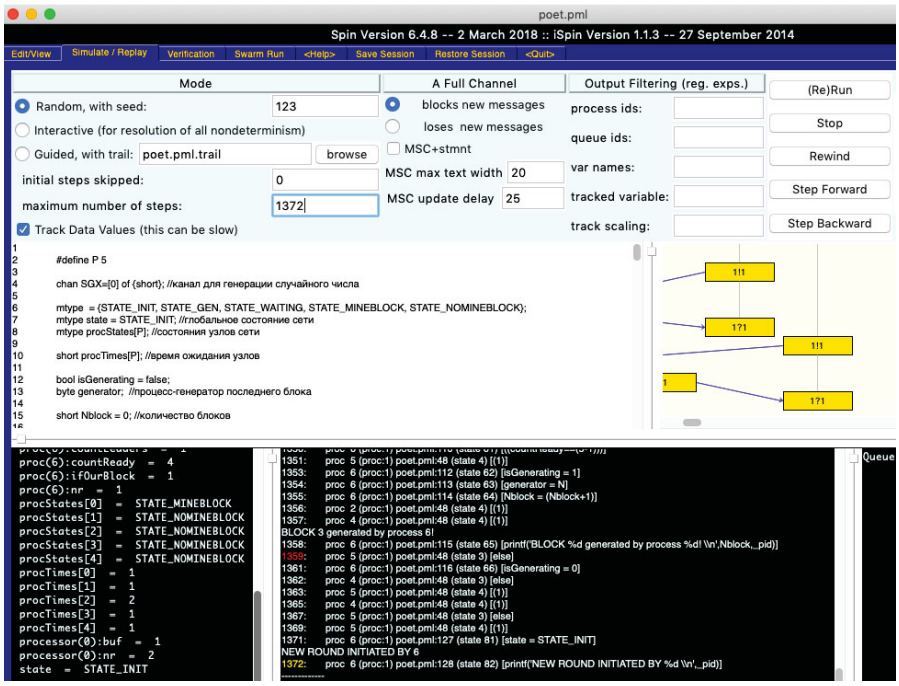

Fig. 5. Simulation of the model with iSpin tool

```
mtype state = STATE_INIT; /* global
    network state */
mtype procStates[P]; /* status of network
     nodes */

short procTimes[P]; /* the waiting time
    of nodes */

bool isGenerating = false; /* during
    block generation will be true */
byte generator; /* process-generator of
    the last block */

short Nblock = 0; /* number of blocks */
```

As for the main process, we implement the algorithm for selecting a leader, based on its description. The expectation, in this case, is modeled by a cycle at a given time. Global variables are used to send global state changes. Each process keeps its local state and can become a leader after the waiting. However, the numbers may match and two or more processes may think that they are leaders, although only one can generate a block. In the model, this is solved by checking the number of leaders according to their states and assigning a reelection procedure in this case. If the formal precondition for generating a block is met, the block is generated and the selection of a new generator is repeated. The model code for the node is shown in Appendix A.

To start the processes, we use the main process and set initial states of the processes and the network.

```
active proctype main() {
  state = STATE_WAITING;
  short count = P - 1;
  do
    :: (count >= 0) -> {
      procStates[count] = STATE_INIT;
          run poet(count);
          count--;
        }
        :: else -> break;
  od
  state = STATE_INIT;
}
```

The simulation of the model using the iSpin tool (a Tcl/Tk simple GUI for SPIN) is shown in Fig. 5. We can see that the network is in a normal state and a leader was elected.

The internal automaton of the model is shown in Fig. 6. Note that the automaton has potentially infinite paths (reelection after reelection) and technologies like state hashing should be used to facilitate verification this model.

To verify the protocol model, we formulate a requirement using a temporal logic expression. Specifically, it states that whenever a block is generated and the L process is designated as the leader, two conditions must hold: the leader must exhibit the minimum waiting time and can only be in the state of block generation. Meanwhile, all other processes should be in a state that indicates they have detected that the block was produced by another entity. The rule can be expressed in a logical form:

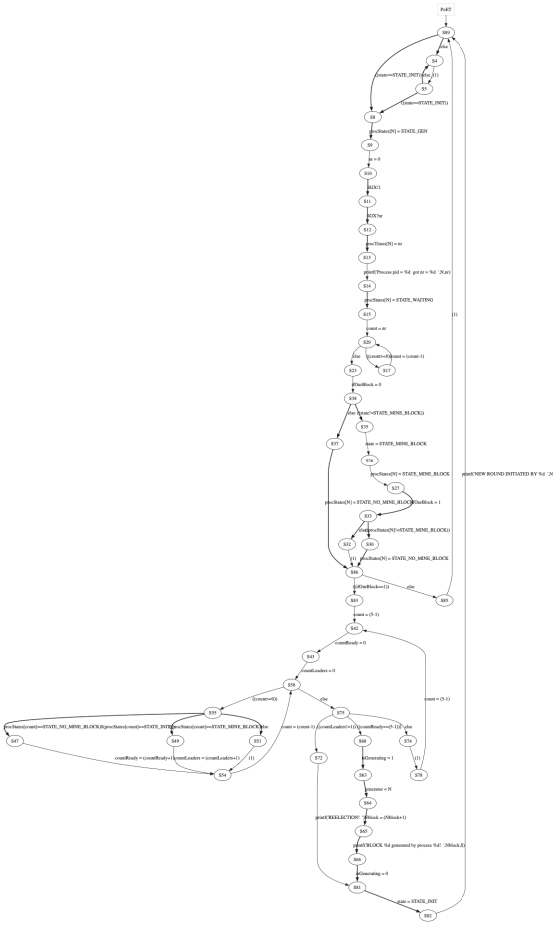

Fig. 6. Internal automaton of the PoET process model: the SPIN tool generates it from the Promela code

$$\forall L \in P : \mathbf{G}((isGenerated \wedge generator == 0) \implies \bigwedge_{\forall I \in P, I \neq L} (procTimes[L] \leq procTimes[I]) \wedge (procStates[L] == STATE\_MINE\_BLOCK) \bigwedge_{\forall I \in P, I \neq L} (procStates[I] == STATE\_NO\_MINE\_BLOCK)) \quad (1)$$

(where **G** is the Globally LTL operator, P is the process set in the model, L is the process number for a current leader, other variables and states were defined above), and then that requirement for the verification purposes is translated into the series of LTL formulas in Promela syntax like this one for the process 0 of 5 total processes in the model:

```
ltl checkFor0 {
[] (isGenerating && (generator == 0) ->
 (procTimes[0] <= procTimes[1] &&
    procTimes[0] <= procTimes[2] &&
    procTimes[0] <= procTimes[3] &&
    procTimes[0] <= procTimes[4]) &&
(procStates[0] == STATE_MINE_BLOCK &&
   procStates[1] == STATE_NO_MINE_BLOCK
   && procStates[2] ==
   STATE_NO_MINE_BLOCK && procStates[3]
   == STATE_NO_MINE_BLOCK && procStates
   [4] == STATE_NO_MINE_BLOCK))
};
```

However, if we try to check LTL properties like this: *a given node will always be able to generate a block at some point*, Model Checker will give a negative answer with a counter-example. Also, properties like *the re-voting process will always end* are not met in our model. Since the protocol is probabilistic, such processes will end in a real system, whereas in the strict model the liveness of the protocol is not guaranteed.

V. Testing Methods for Enterprise Blockchain Applications

Earlier we described the general architecture of Hyperledger Sawtooth. For a convenient and quick launch enterprise blockchain applications under tests is proposed to containerize all Hyperledger Sawtooth nodes.

Consider the diagram in Fig. 1. For this architecture, it is possible to implement a single description file using the Docker-compose syntax, which allows us to deploy the entire

environment with a single command, so it can be useful during Unit- or Integration Testing, as well as in Continuous Integration / Deployment practices. At the same time, tests for transaction processors can be run both inside and outside of containers.

Transaction processors themselves may not be turned into a container, but only connected to a validator inside a Docker network, if it is necessary to run the tests repeatedly locally, for example on the developer's PC.

The architecture of Hyperledger, along with its API and containerization features, promotes innovative programming practices such as Test-Driven Development (TDD) [33]. In TDD, development begins by writing tests that reflect the initial requirements. Next, developers create minimal stub code to meet those requirements. As time allows, developers can refine this code by introducing more sophisticated tests and enhancing the underlying logic (see how to teach/learn it in [24]). Notably, Hyperledger's design simplifies the development process, allowing developers to focus on testing and infrastructure setup with just a command, without needing to worry about the complexities of network organization.

However, the essence of this architecture is ephemeral, that is, their operation results can not be saved in any way, also, node keys are generated during the startup process. So it is extremely problematic to get access to them, but they are needed when a blockchain application is implemented, since the transactions of which should be signed with the keys known in advance. In this case, we can use a Docker volume that is bound to a physical folder in the host OS. Note that it is necessary to connect it both to the validator and the container with the consensus engine (in particular, the PoET engine), but the data mapping must be made to different folders. This is schematically shown in Fig. 7.

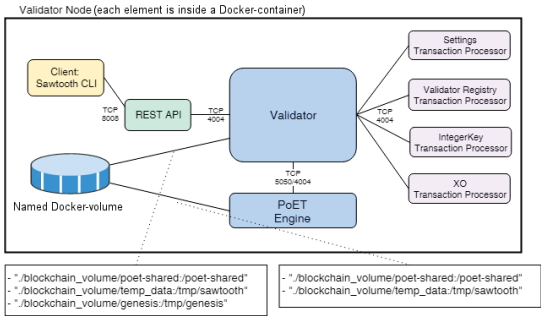

Fig. 7. Architecture of a Hyperledger Sawtooth node validator with a mounted shared data volume

However, the above does not allow us to create an infrastructure that we can restart, update the code of its elements (transaction preprocessors), use its state in other processes. It can be useful for the final stage of testing (manual) or some end-to-end tests. We propose to run such infrastructure in three scenarios:

- Generation of the keys and basic configuration files. With the help of console utilities of the Hyperledger Sawtooth validator, a required number of public and private keys are generated in advance, they are stored in the Docker volume mounted to the host OS folder. Basic configuration files are also generated based on these keys.
- Generation of genesis-block. With the help of console utilities of the Hyperledger Sawtooth validator, previously obtained private keys, derived configuration files, the so-called genesis-batch is generated – this is the first transaction in the blockchain, in which some rules of behavior of the network participants of this blockchain are written. This item is also stored in a special folder on the mounted volume.
- Starting peer nodes without genesis.
- Running the overall infrastructure. The blockchain infrastructure is created on the basis of genesis-batch and keys.

Let us revisit the issue of backing up blockchain data for future use. Hyperledger Sawtooth utilizes what are known as sparse files to maintain the blockchain state. While the system indicates that the space required is over 1TB, the actual size may only be a few kilobytes. This discrepancy presents a problem: mounting a volume in a folder on the host operating system is inadequate if the OS does not have more than 1TB of available space. However, this issue can be resolved by using a different type of Docker volume that is not directly linked to the host OS folder, allowing for physical backups without requiring substantial disk space. The connection of such a volume is shown in Fig. 8.

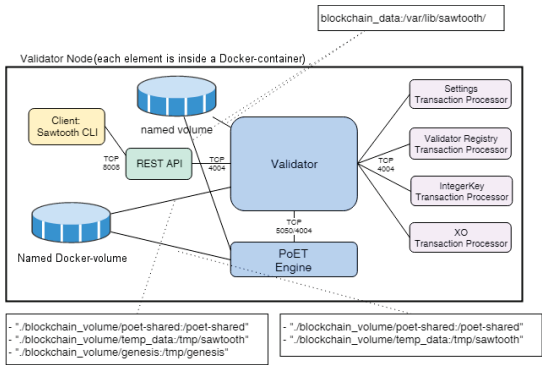

Fig. 8. Final architecture of our Hyperledger Sawtooth node validator

Now, let us focus on the development of the Ending Transaction Processor (TP), which was specifically created for this project. Currently, the Sawtooth framework has a notable behavior: if the last transaction within a received packet results in an error (a scenario that can arise when testing the functionality of a transaction processor), the validator continuously rebroadcasts this erroneous transaction until it processes a new successful one. This behavior negatively affects the accuracy of metrics and logs associated with the tested components. To address this issue, we have designed a specialized transaction processor that adds a new element to the transaction package. This new transaction will not only be guaranteed to succeed but can also include various metadata related to the current test or the entire testing suite. This enhancement facilitates more comprehensive analytics moving forward.

VI. Conclusions

As a result of our research on Hyperledger Sawtooth, we successfully integrated two highly beneficial methodologies: formal verification to ensure the integrity of the PoET consensus mechanism and Docker to implement CI/CD practices in the development of enterprise blockchain applications.

We demonstrate the application of the Model Checking technique to represent the PoET consensus in a simplified network environment. The insights gained from this modeling can be leveraged to develop more complex models to rigorously check protocol behavior.

Additionally, employing container virtualization for the components of Hyperledger Sawtooth has enabled the incorporation of various DevOps practices into the development lifecycle of Blockchain 3.0 applications on this platform. This integration has had a favorable impact on the quality of our output. The effective containerization of nodes has significantly expedited both the development process and the deployment of the final blockchain system. Moreover, the transaction processor we developed has been successfully implemented in a production environment, providing enhanced logging and metrics that align with our objectives.

Acknowledgement

The author would like to express his sincere gratitude to Roman Galkin from WinteX Solutions for his work on a master's thesis on a related topic.

References

[1] J. Abou Jaoude and R. G. Saade, "Blockchain applications–usage in different domains," *IEEE Access*, vol. 7, pp. 45 360–45 381, 2019.

[2] Q. Wang, R. Li, Q. Wang, S. Chen, M. Ryan, and T. Hardjono, "Exploring web3 from the view of blockchain," *arXiv preprint arXiv:2206.08821*, 2022.

[3] Z. Zheng, S. Xie, H.-N. Dai, W. Chen, X. Chen, J. Weng, and M. Imran, "An overview on smart contracts: Challenges, advances and platforms," *Future Generation Computer Systems*, vol. 105, pp. 475–491, 2020.

[4] M. Zachariadis, G. Hileman, and S. V. Scott, "Governance and control in distributed ledgers: Understanding the challenges facing blockchain technology in financial services," *Information and organization*, vol. 29, no. 2, pp. 105–117, 2019.

[5] A. Sharma, F. M. Schuhknecht, D. Agrawal, and J. Dittrich, "How to databasify a blockchain: the case of hyperledger fabric," *arXiv preprint arXiv:1810.13177*, 2018.

[6] D.-J. Munoz, D.-A. Constantinescu, R. Asenjo, and L. Fuentes, "Clinicappchain: A low-cost blockchain hyperledger solution for healthcare," in *International Congress on Blockchain and Applications*. Springer, 2019, pp. 36–44.

[7] C. Banks, S. Kim, M. Neposchlan, N. Velez, K. Duncan, J. James, A. St Leger, and D. Hawthorne, "Blockchain for power grids," in *IEEE SoutheastCon*, 2019.

[8] N. Gaur, L. Desrosiers, V. Ramakrishna, P. Novotny, S. A. Baset, and A. O'Dowd, *Hands-On Blockchain with Hyperledger: Building decentralized applications with Hyperledger Fabric and Composer*. Packt Publishing Ltd, 2018.

[9] S. Staroletov and R. Galkin, "Testing methods for blockchain applications," in *International Conference on High-Performance Computing Systems and Technologies in Scientific Research, Automation of Control and Production*. Springer, 2021, pp. 401–418.

[10] C. Cachin and M. Vukolić, "Blockchain consensus protocols in the wild," *arXiv preprint arXiv:1707.01873*, 2017.

[11] L. Chen, L. Xu, N. Shah, Z. Gao, Y. Lu, and W. Shi, "On security analysis of proof-of-elapsed-time (poet)," in *International Symposium on Stabilization, Safety, and Security of Distributed Systems*. Springer, 2017, pp. 282–297.

[12] Y. Shen, Y. Chen, K. Chen, H. Tian, and S. Yan, "To isolate, or to share?: That is a question for Intel SGX," in *Proceedings of the 9th Asia-Pacific Workshop on Systems*. ACM, 2018, p. 4.

[13] M. Schwarz, S. Weiser, and D. Gruss, "Practical enclave malware with Intel SGX," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2019, pp. 177–196.

[14] A. Miller, "Permissioned and permissionless blockchains," *Blockchain for distributed systems security*, pp. 193–204, 2019.

[15] X. Chen, S. He, L. Sun, Y. Zheng, and C. Q. Wu, "A survey of consortium blockchain and its applications," *Cryptography*, vol. 8, no. 2, p. 12, 2024.

[16] V. Dhillon, D. Metcalf, and M. Hooper, "The hyperledger project," in *Blockchain enabled applications*. Springer, 2017, pp. 139–149.

[17] *Hyperledger projects*. [Online]. Available: https://www.hyperledger.org/projects

[18] K. Olson, M. Bowman, J. Mitchell, S. Amundson, D. Middleton, and C. Montgomery, "Sawtooth: An introduction," *The Linux Foundation, Jan*, 2018.

[19] H. M. Sayeed, M. Abu-Amara, and H. Abu-Amara, "Optimal asynchronous agreement and leader election algorithm for complete networks with byzantine faulty links," *Distributed Computing*, vol. 9, no. 3, pp. 147–156, 1995.

[20] F. S. Gharehchopogh and H. Arjang, "A survey and taxonomy of leader election algorithms in distributed systems," *Indian journal of science and technology*, vol. 7, no. 6, p. 815, 2014.

[21] Hyperledger, "Sawtooth-poet," 2019, https://github.com/hyperledger/sawtooth-poet.

[22] Intel, "Intel® software guard extensions sdk for linux os," 2016. [Online]. Available: https://01.org/sites/default/files/documentation/intel_sgx_sdk_developer_reference_for_linux_os_pdf.pdf

[23] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient Out-of-Order execution," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 991–1008.

[24] S. Staroletov, "Teaching the discipline "Software testing and verification" to future programmers," *System Informatics*, no. 21, pp. 1–28, 2022. [Online]. Available: https://system-informatics.ru/files/article/n21-staroletov.pdf

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

[26] N. O. Garanina, I. S. Anureev, V. E. Zyubin, S. M. Staroletov, T. V. Liakh, A. S. Rozov, and S. P. Gorlatch, "A temporal logic for programmable logic controllers," *Automatic Control and Computer Sciences*, vol. 55, no. 7, pp. 763–775, 2021.

[27] S. Staroletov and N. Shilov, "Applying model checking approach with floating point arithmetic for verification of air collision avoidance maneuver hybrid model," in *F. Biondi et al. (Eds.): SPIN 2019, LNCS 11636*. Springer Nature Switzerland AG, 2019, p. 15.

[28] Nimble-code, "Dolev, Klawe & Rodeh for leader election in unidirectional ring," 2015. [Online]. Available: https://github.com/nimble-code/Spin/blob/master/Examples/leader0.pml

[29] M. Virmani, "Understanding devops & bridging the gap from continuous integration to continuous delivery," in *Fifth International Conference on the Innovative Computing Technology (INTECH 2015)*. IEEE, 2015, pp. 78–82.

[30] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Adobe Reader)*. Pearson Education, 2010.

[31] S. Staroletov, "PoET model source code in Promela." [Online]. Available: http://github.com/SergeyStaroletov/PromelaSamples/blob/master/PoET.pml

[32] Promela, "Rand - for random number generation," http://spinroot.com/spin/Man/rand.html.

[33] K. Beck, *Test-driven development: by example*. Addison-Wesley Professional, 2003.

APPENDIX A
MODELING POET IN THE PROMELA FORMAL LANGUAGE

```
proctype poet(byte N) {
do
:: {
    /* waiting for a next iteration -- for state STATE_INIT */
        do
            :: (state == STATE_INIT) -> break;
            :: else -> skip;
        od

        procStates[N] = STATE_GEN;
        /* generate a random number by asking it from the special process */
        short nr = 0;
        SGX ! 1;
        SGX ? nr;
        procTimes[N] = nr;
        printf("Process␣pid␣=␣%d␣␣got␣nr␣=␣%d␣\n", _pid, nr);
        procStates[N] = STATE_WAITING;

        /* simulate the waiting: in the loop we decrement count */
        short count = nr;
        do
            :: (count >= 0) -> count--;
            :: else -> break;
        od

        /* after the waiting we  check for the block present and try to generate it */
        bool ifOurBlock = false;
        if
            :: (state != STATE_MINE_BLOCK) -> {
                /* if not, we are the first and we name us the leader */
                atomic {
                    state = STATE_MINE_BLOCK; /* we mark the the block is mined */
                    procStates[N] = STATE_MINE_BLOCK; /* mark that the block is ours */
                    ifOurBlock = true;
                }
                if  /*otherwise the block is not ours */
                    ::(procStates[N] != STATE_MINE_BLOCK) ->
                    procStates[N] = STATE_NO_MINE_BLOCK;
                    ::else -> skip;
                fi
            }
            :: else -> procStates[N] = STATE_NO_MINE_BLOCK;
        fi
        /* next there is the logic of block generation */
        if
            :: ifOurBlock == true -> {
                /* we think we are the leader - wait for other processes */
                do
                :: {
                    count = P - 1;
                    short countReady = 0;
                    short countLeaders = 0;
                    do
```

```
                        :: (count >= 0) -> {
                            if
                                /* calculate count of non-leader processes */
                                :: (procStates[count] == STATE_NO_MINE_BLOCK) ||
                                /* and also new processes */
                                (procStates[count] ==  STATE_INIT) -> {
                                    countReady++;
                                }
                                /* count the leaders */
                                :: (procStates[count] == STATE_MINE_BLOCK) ->
                                countLeaders++;
                                :: else ->  skip;
                            fi
                            count--;
                        }
                        :: else -> break;
                    od
                    if :: (countReady == P - 1) ->
                    /* normal state: I am the leader and there are no others */
                    {
                        isGenerating = true;
                        generator = N;
                        Nblock++;
                        printf("BLOCK␣%d␣generated␣by␣process␣%d!␣\n", Nblock, _pid);
                        isGenerating = false;
                        break;
                    }
                    :: (countLeaders != 1) -> {
                    /* something wrong: more than one leader, reelection */
                        printf("REELECTION!␣\n");
                            break;
                    }
                    :: else -> skip;
                    fi
                } od
                /* initiate a new round */
                printf("NEW␣ROUND␣INITIATED␣BY␣%d␣\n", _pid);
            }
            ::else ->
            skip;
        fi
    }
od
}
```