Research Article

Formal Modelling of PBFT Consensus Algorithm in Event-B

Jie Li,1,2 Kai Hu,1,2 Jian Zhu,3 Jean-Paul Bodeveix,4 and Yafei Ye5

1State Key Laboratory of Software Development Environment, Beihang University, Beijing, China
2Key Laboratory of Blockchain Application Technology of Yunnan Province, Yunnan, China
3The 14th Research Institute of CETC, Nanjing, Jiangsu, China
4Institut de Recherche en Informatique de Toulouse, Toulouse, France
5Yunnan Innovation Institute of Beihang University, Yunnan, China

Correspondence should be addressed to Yafei Ye; yeyf@bhynii.com

Received 7 January 2022; Revised 10 July 2022; Accepted 29 August 2022; Published 14 October 2022

Academic Editor: Jinguang Han

Copyright © 2022 Jie Li et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

The practical Byzantine Fault Tolerance (PBFT) is a classical consensus algorithm that has been widely applied in an alliance blockchain system to make all nodes agree to certain transactions under the assumption that the proportion of Byzantine nodes is no more than 1/3. It is prevalent due to its performance, simplicity, and claimed correctness. However, any vulnerability of the consensus algorithm can lead to a significant loss in finance because no one can change the transaction results after execution. This paper proposes a formal development method of the PBFT algorithm by horizontal refinement in Event-B, which allows us to manage the complexity of the proof process by factoring the proof of correctness into several refinement steps. During the development of PBFT, we have specified the core mechanism like parameterized message types, primary node change, and water-mark interval. Furthermore, we present a mechanical verification of the safety and liveness properties of the model in Rodin, which can be partially and widely used to check the blockchain consensus algorithm vulnerability using a refinement tree of algorithms.

1. Introduction

Blockchain technology is an emerging decentralized distributed system formed by using encryption algorithms, P2P technology, tree structure, consensus algorithm, reward mechanism, etc. It can be argued that blockchain is a distributed ledger with the characteristics of decentralization, immutability, traceability, trust, openness, and transparency. Due to these characteristics, blockchain technology has been widely applied in many industrial fields such as finance, cloud computing, and the Internet of Things. Bitcoin [1], as a representative of the first generation of blockchain technology, provides people with popular virtual currencies. It has successfully solved the double-spending problem and the Byzantine faults. Ethereum serves as the symbol of the second generation of blockchain while providing smart contract functionality. With the success of Bitcoin and Ethereum, there is more and more research on blockchain technology. As the core of blockchain technology, the consensus mechanism fundamentally determines the security, availability, and system performance of the entire blockchain system. Research on the consensus mechanism of the blockchain is of great significance to the expansion of the blockchain, the increase of transaction processing speed, and the improvement of security.

Currently, the consensus mechanism is mainly evaluated from four dimensions: security, scalability, efficiency, and resource consumption. Security is an essential point. A consensus algorithm ensures that all node operations are consistent with the service specifications and that no undesirable results occur during the process. Furthermore, we consider that it has good fault tolerance and can detect Byzantine errors, etc.

According to research [2], almost all the distributed systems which support strong consensus use Byzantine fault-tolerant protocols as their core algorithm. PBFT (Practical Byzantine fault tolerance) serves as one of their main building blocks or inspires them, such as [3–5]. Besides, compared to the proof of work (POW) [6] algorithm and the proof of stake (POS) [7] algorithm, the PBFT [8] is more prevalent in
Consortium Blockchain because it consumes less computation power. Even though the safety of the PBFT has been mathematically and manually proved, formal verification of safety and liveness is still concerned with confirming the distributed system consistently runs. Furthermore, if a typical alliance chain uses PBFT as the consensus protocol, some details may also be modified and need to be verified again. In this case, the model will still be instructive, and sometimes only a few configurations have to be redefined.

There are a lot of formal verification tools and research about verifying protocols, as shown in Section 2. Event-B [9] is a formal modeling language that supports specifying and implementing algorithms and systems as discrete transition systems based on a typed set theory. Since it integrates some tools which can verify the smart contracts or verify the liveness property automatically and manually proved, formal verification of PBFT is the most popular theorem proving tools, and ProB is a model checker for the B-Method, which can be called to check specific errors.

We use horizontal refinement and ProB plug-in to build the protocol incrementally with the Event-B language and the tool Rodin platform [10]. Then these refinements are introduced to make Rodin produce the wanted proof obligations for stabilization. The stabilization property is a particular kind of liveness property. For any specific instance, all the messages sent from clients for execution will be confirmed by nodes ultimately, which means all nodes will reach a consensus. We rely on Event-B and Rodin to take advantage of the ability to write specifications in an expressible language that the built-in generator can translate and forward to SMT (Satisfiable Modulo Theories) solvers, which is logical first-order formulas of a decision problem. Our model illustrates the core mechanism of the PBFT, while different application scenarios may lead to tiny differences in details.

The main contributions of this paper are as follows:

1. Detailed specification and formal modeling of PBFT by horizontal refinement implement critical features like weak synchrony, view change functions, and watermark interval. In addition, we provide a generic model, which can implement various specific protocols.

2. Formal verification of the agreement property by proving the property that any message being effectively agreed on should have at least f + 1 identical execution result from different nodes.

3. We apply the weak fairness theory to verify the liveness property. A suitable variant is proposed to prove that all convergent events should remain globally enabled and decrease the variant. In contrast, the others should not increase it.

In Section 2, we present the description of Event-B and the PBFT as a preliminary. In Section 3, some related work about formal verification of protocol algorithm is given. In Section 4 and 5, we address the development of the PBFT by stepwise refinement and verify the agreement property. In Section 6, we prove the expected liveness property under a weak fairness assumption. Finally, we give a short analysis and summary of the development as the conclusion and discuss future work in Section 6 and discuss future work in Section 7.

2. Preliminary


(1) A context comprises abstract sets that define data types and constants linked to some properties defined as axioms.

(2) A machine contains variables, invariants, theorems, variants, and events. A machine has variables associated with invariants and events. An event consists of a guard and an action. The guard denotes the enabling condition of the event, and the action represents how the event modifies the state of the machine. Values of machine variables represent different machine states.

As shown in Figure 1, a machine can refine another machine and inspect one or more contexts. Contexts can extend to other ones as well.

2.2. The Process of PBFT. As mentioned above, PBFT was proposed by Miguel Castro and Barbara Liskov in 1999 [8] to solve the Byzantine problem in asynchronous distributed systems. Their model adds the design of timeout for bypassing the impossibility of FLP [11], which proved no algorithm could guarantee the consensus in asynchronous and distributed systems, and hence it is called weak synchronization. For the default transmission of information, nodes use cryptographic technology to sign the information to prevent malicious behavior from tampering with the message. There are some restrictions on the number of nodes. We assume that most f nodes may exhibit Byzantine failure, and most other f nodes may exhibit crash failures. To ensure safety and liveness, responses from nonfaulty nodes must outnumber faulty nodes, which means n − 2f > f. Therefore n > 3f (n represents the number of total nodes).

The standard process of the algorithm is mainly divided into five steps: Request, Pre-Prepare, Prepare, Commit and Reply stages. According to these steps, the messages have five categories: REQUEST, PRE-PREPARE, PREPARE, COMMIT, and REPLY. In addition, nodes in the system contain primary nodes and backup nodes (except the primary node). The five processes are as follows:

1. Request stage

The client c sends a request message Request (o,t,c) to the primary node, where o is the request by client c and t is a timestamp marking the execution time of the client. If the client sends the message to a backup node, the node will forward it to the primary node.

2. Pre-prepare stage

The primary node will check and broadcast the request message to all the other nodes. The nodes receive and verify
the message from the primary node. After the verification, they will broadcast the prepare message \((v,n,d,m)\). Here, \(v\) represents the view in which the message is being sent, \(n\) indicates the sequence number assigned by the primary node for ensuring the consistency of non-Byzantine nodes in the system, \(m\) is the client’s request message, and \(d\) is \(m\)’s digest;

(3) Prepare stage

The backup nodes check the soundness of the received Pre-Prepare message. After the verification, the message Prepare \((v,n,d,i)\) is sent to all other nodes (including the primary node). Here, \(i\) is the identity of the node;

(4) Commit stage

The primary node and the backup node verify the soundness of the received Prepare message. If the node receives 2\(f\) + 1 valid messages, it will send the Commit \((v,n,d,i)\) message to the other nodes (including the primary node);

(5) Reply

The node verifies the soundness of the received COMMIT message. If the node receives 2\(f\) + 1 valid messages, it allows the client's request to operate 'o', and returns Reply \((v,t,c,d,r)\) to the client. In this format, \(r\) represents the result of the requested operation. If the client receives \(f\) + 1 identical Reply message, it means that the request initiated by the client has reached the consensus of the entire network.

Figure 2 shows the normal process where the primary node is nonfaulty [8]. In detail, \(C\) is the client, node 0 is the primary node (not faulty), and node 3 crashes and does not respond to the received messages.

When the primary node fails, a view-change is needed to ensure the liveness of the distributed system, as shown in Figure 3. The timer of the backup node ends view \(v\) due to a long wait. The view-change mechanism is triggered to change the system to view \(v + 1\). The node will broadcast the message ViewChange \((v + 1,n,C,P,i)\) to the entire system, and the node will only accept checkpoint, view-change, and new-view message types. Here, \(i\) is the node’s identity, \(n\) equals the latest stable checkpoint \(s\) known to \(i\). \(C\) is a set containing 2\(f\) + 1 valid checkpoint messages for proving the correctness of checkpoint \(s\). \(P\) is a set that includes the information of the Pre-prepare and Prepare messages with a sequence number higher than \(n\).

When the primary node \(P\) of view \(v + 1\) receives 2\(f\) valid view-change messages, it will broadcast the message NewView\((v + 1,V,O)\) to other nodes, where \(V\) is the set of received view-change messages (including the message sent by itself), and \(O\) contains Pre-prepare messages to be confirmed. After receiving the new-view type message, the backup node verifies the content of the view \(v + 1\) and the correctness of the set \(O\). If the message is valid, the node will enter view \(v + 1\).

3. Related Work

Consensus mechanism serves as the fundamental and core technology of blockchain. It determines how the participating nodes agree on certain specific data. Consensus agreements have been studied for a long time, and the consensus mechanisms contain classic distributed and blockchain ones.

As early as 1975, Akkoyunlu et al. [12] put forward the “two armies’ problem” in the computer science field, which opened the research on consensus mechanism. In 1978, Lamport defined a logical clock system to sequence events and introduced a state machine replication method in a distributed system to solve the mutual exclusion problem of resources in [13]. It is the first work involving consensus in distributed systems. In 1982, Leslie Lamport et al. [14] proposed the original presentation with the basic protocols “Oral Messages” (OM) and “Signed Messages” (SM) to solve the Byzantine General Problem. However, the cost of these solutions is very high and requires \(O(N^2(f+1))\) for the number of Byzantine nodes in information exchange. In 1989, Leslie Lamport proposed the Paxos [15] algorithm based on the process of passing legislation in Congress without considering the Byzantine failure. This algorithm contains three roles: proposer, reviewer, and learner; it is divided into two stages: proposal preparation (consensus discussion) and proposal release (consensus release). But it does not support Byzantine Fault Tolerance.

In 1999, Miguel Castro et al. proposed PBFT to the Byzantine Generals question [8]. In [11], Fischer, Lynch, and Paterson proposed the impossibility of FLP which proved that no algorithm could guarantee the consensus in an asynchronous system, even if only one process failed. For bypassing the impossibility of FLP, this algorithm used synchronization assumptions. And it reduced the cost of information exchange to \(O(n^2)\) (\(n\) is the number of nodes) as well.

In 2008, Satoshi Nakamoto used POW in Bitcoin and the consensus mechanism enters the era of blockchain consensus. POW indicated each node in the system conducts a round of computing power every ten minutes and the winner node...
obtains the right to keep accounts and synchronizes the new additions to other nodes. Due to the POW algorithm, the whole bitcoin system consumes excessive resources every year. In 2012, the Quantum Mechanic POS [7] consensus algorithm was proposed and used in the Peercoin system. POS indicated the node with the highest equity obtains the right to keep accounts. Compared with POW, POS has the advantage of less resource consumption. But it was easy to bring the centralization problem. In 2013, the Bitshares project proposed DPOS (Delegated proof-of-stake) [16] based on POW and POS, in which each node voted to form a board of directors for consensus. In 2014, Diego Ongaro [17] proposed the Raft algorithm. It inherited the advantages of POS and weakened the centralization problem. The algorithm contains three roles: leader, candidate, and follower. To achieve easy-to-understand purposes, the entire consensus process is simplified into four independent subproblems: leader election, log replication, safety and membership change, and resolved separately. However, the Raft algorithm cannot support a Byzantine system, which is likely more used in Blockchain.

According to [2], almost all distributed systems which support strong consensus use BFT protocols as the core algorithm, and PBFT is one of most classic, representative BFT algorithm. Compared with previous BFT algorithm, PBFT supports higher fault tolerance and it reduces the complexity of the algorithm to \(O(n^2)\).

PBFT is not only an excellent consensus algorithm which solves the traditional distributed consensus problems, but also becomes the subject of more and more formal methods for modelling and verification. For example, a new state-machine replication algorithm is used to prove PBFT’s safety and liveness by describing asynchronous distributed systems [18] and stating their properties. Velisarios [19] is used to model and verify the safety of asynchronous Byzantine fault-tolerant protocols using Coq. An Event-B model [20] is used to prove the agreement and validity of a Byzantine agreement algorithms ZA and SM. Heard-Of model was implemented in Isabelle/HOL [21]. The work [22] investigates the current research on the use of formal methods to verify consensus protocols and believes model checking is used in the majority. However, most studies have focused on proving safety, sometimes with manual proofs only, while the correctness property for PBFT contains both safety and liveness properties in our work.

Other consensus mechanisms have been formally studied using "high-level" specification languages. For instance, TLA+ has been used to prove safety and liveness of Multi-Paxos [23] and safety of a variant of an abstract model of PBFT [24]. PSync [25] is a domain specific language based on the HO-model, and is used to implement crash fault-tolerant algorithms. ByMC is counter-based model checker used to verify safety and liveness of fault-tolerant distributed algorithm [26–28].

Compared with the previous studies, we use horizontal refinement to build the PBFT incrementally. Then technical refinements are introduced to make Rodin produce the wanted proof obligations for correctness. Our work is the first to verify both the safety and liveness property of PBFT in a parameterized setting and use a much automated, mechanical proof.

4. Requirements for Models

Before developing a PBFT, we should consider the environment assumptions and the requirements for the constructed model. The protocol works in a network environment where the communications among the client, the primary node, and all other nodes are asynchronous and unbelievable. For example, messages can be dropped, altered, delayed, duplicated, or delivered out of order. Therefore, we need to
put forward more precise and appropriate requirements to ensure the correctness of the model. The refined model should safely and reliably run in such a complex environment. After carefully analyzing the algorithm, we propose the assumptions and the system requirements that we make about the various components of the algorithm:

(i) A1. The network consists of a finite set of nodes and clients forming a P2P architecture

The network is made up of a large but finite number of nodes, because the more distributed nodes participate in the consensus, the more reliable the system is. Considering the actual situation and the time complexity of the algorithm, the number of nodes cannot be infinite. These nodes are positioned on different sites that are connected in a distributed network architecture as indicated in Figure 4.

(ii) A2. Each node can send a message to all other nodes in the network and receive messages sent by any node

(iii) A3. Messages can be buffered in each node

As illustrated in Figure 5, all messages received by the node will be stored in a buffer which is associated with the node. In particular, messages that have been processed and

---

**Figure 6: Model Development.**

**Figure 7: Initial model.**
messages that are waiting for consensus will be placed in the buffers in order.

(iv) A4. Messages are reliable

A node (even a faulty one) cannot forge messages. It is up to the user signature and transfer methods to ensure that this is indeed the case. Usually, this means that forged messages can be created but are invalid (e.g. digital signatures violated) and, hence, forgery can always be detected.

(v) R1. For the same message, the results executed by all correct nodes are the same

If a message is executed by a CORRECT node, the result obtained should also be correct and unique. It guarantees the liveness and the safety of the system.

(vi) R2. If at most $f$ nodes are faulty which means that they can send wrong execution results, and at most $f$ nodes have crashed which means that they cannot forward messages, there should be at least $3f + 1$ nodes in total to ensure the reachability of a consensus

(vii) A5. The model is weak synchronous

We suppose that a peer node responds within a specific time, and this specific duration will not change. Messages may be discarded in the network, delayed, and arrive out of order.

(viii) A6. The number of messages waiting to be processed cannot exceed a certain threshold

This assumption reflects the setting of water mark interval. The client will stop sending new transactions when $k$ messages are waiting to be processed, and it will continue to send messages when the number of messages waiting to be processed is less than $k$.

(ix) R3. All nodes may crash or go evil. When the primary node crashes, another correct node will be chosen to act as the primary node

This requirement guarantees the problem of view change in the PBFT and states that any node in the model can evolve from correct to faulty.

(x) A7. Each node has a unique number

We number all nodes, and if the primary node needs to be changed when it crashes, we can simply choose the next one as the new primary node.
(xi) R4. A consensus process needs to go through five steps

5. Development

We now introduce the verification of the PBFT. As shown in Figure 6, we illustrate four steps of developing a formal model for the PBFT. Each model contains a machine and a context. The initial machine specifies the state of any successful run of the PBFT consensus algorithm where we prove the agreement and liveness properties, then M1 refines M0 with the concrete message type and the water mark interval definition. Next, M2 refines M1 with the complete consensus process and M3 refines M2 with more specific false behaviors of nodes.

Our initial model is very abstract compared to the PBFT. It only contains three stages: request, commit, and reply. And there are three kinds of messages. The request message and commit message has two items: Type and Content. Type is the set containing the three message types: request, commit, and reply. Content contains the effective contents of a message in PBFT. The reply message has an additional item: Result, which indicates the execution’s result of the operation requested by the client. Figure 7 describes the state of the nodes and the system. The part of “State of node” focuses on the state after the primary node broadcasts the request message. And three events are needed: commit, reply and Go_faulty. In the commit event, the primary node broadcasts the message to all nodes in the distributed system. In the reply event, a node executes the operation requested by the client. The event Go_faulty specifies the situation where the normal node becomes faulty during the commit and reply phases. The event Go_faulty only affects the final reply process on the contents of the message. In Figure 7, the message marked in red is caused by the Go_faulty event. The part of "State of system" additionally contains the request and primary_change event. In the request event, the client sends a request to the primary node. In the primary_change event, the system re-selects another node as the primary node.

The liveness property is guaranteed by the event primary_change which is fired when the primary node crashes. In this case, the system selects another node as the primary node so it will not enter a deadlock. The primary node in the system should belong to the set of correct nodes and behaves correctly. Otherwise, the event primary_change will be fired. The proof of the agreement property relies on the fact that the local views of correct nodes are identical at
the end of any execution. To show this, we specify that more than \( f + 1 \) identical results of the same message from the system are regarded as valid responses to the client. Besides, another important thing is that the initial model emphasizes the asynchrony of the process where messages are sent by clients and the process of consensus between nodes.

As shown in Figure 8, the first refinement M1 mainly refines the message by adding the sequence number \( N \) which is assigned by the primary node. Besides, M1 imports the notion of low level of the water mark interval \([h, H]\). In Figure 8, we add the judging conditions: within the interval \([h, H]\) or under low-water, and we limit that all messages processed by nodes cannot exceed \( N \) by a specified value \( H \) which is the high level of the water mark interval. So that we can control the complexity of the entire model by modifying the high level of the water mark interval. When \( H \) is bigger, the model is more complicated for the asynchrony. Meanwhile, there are messages with the sequence number under low water level \( h \), they will be updated automatically with the execution results stored in correct nodes’ buffers.

As shown in Figure 9, The second refinement M2 refines the consensus process of the PBFT in M1. Compared with the previous model, it includes the complete consensus process of PBFT and adds two inspection conditions: In Prepare_check, each node checks if there are \( 2f + 1 \) identical message have been received from different nodes; in

---

**Listing 1: Context of M0.**

```
Sets
Message result
Constants
CORR NODES True_execute FAULTY
Axioms
@axm0 NODES⊂N
@axm1 finite(NODES)
@axm2 NODES≠∅
@axm3 CORR⊂NODES
@axm4 CORR≠∅
@axm5 3·card(CORR) ≥ 2·card(NODES) + 1
@axm6 result≠∅
@axm7 FAULTY∈result
@axm8 True_execute(message)→result\{FAULTY\}
End
```
Commit_check, each node checks if there are \(2f+1\) identical commit messages have been received from different nodes or itself. Because the event Go_faulty could fire during the events Prepare, Commit and Reply during the change of state of the node and system. Finally, the commits for execution could be faulty and the node reply with a wrong execution result of the message.

As shown in Figure 10, The third refinement M3 refines the M2 by importing concrete false behaviors of nodes through the events Go_Evil and Go_Crash. Any node in a crash cannot forward messages to other nodes or the client, and the node will remain in its previous state. Any node that goes evil will forward false messages and reply with wrong execution results. In particular, the primary node cannot go evil; if it does, it will be switched. Which is formulated by requirement R3.

In the following sections, we will introduce the modeling details of the PBFT and explanations of the relevant codes, while some unimportant code details will not be presented.

5.1. The Initial Model. The purpose of the initial model is to define the success conditions for PBFT. This model should be as abstract as possible and specify the critical liveness and agreement properties to make them valid. We begin by defining the initial Event-B context as shown in Listing 1.

In this context, we define a carrier set Message to represent

\[
\begin{align*}
\text{Invariants} & \quad \text{\(\forall i \in \text{NODES} \rightarrow \text{msg} \rightarrow \text{result} \)} \\
\text{inv1} & \quad \text{\(\text{G} \in \text{NODES} \rightarrow \text{message} \rightarrow \text{result} \)} \\
\text{inv2} & \quad \text{\(\text{partition}((\text{NODES}, \text{Faulty}, \text{corr})) \)} \\
\text{inv3} & \quad \text{\(3 \times \text{card}((\text{corr})) \geq 2 \times \text{card}((\text{NODES})) + 1 \)} \\
\text{inv4} & \quad \text{\(\text{cache} \rightarrow \text{message} \)} \\
\text{inv5} & \quad \text{\(\forall i \in \text{corr} \rightarrow \text{dom}(G(i)) \subseteq \text{cache}(i) \)} \\
\text{inv6} & \quad \text{\(\text{pre} \rightarrow \text{NODES} \)} \\
\text{inv7} & \quad \text{\(\text{Pre_set} \subseteq \text{NODES} \)} \\
\text{inv8} & \quad \text{\(\forall n \in \text{corr} \rightarrow G(n) \subseteq \text{True_execute} \)} \\
\text{inv9} & \quad \text{\(3 \times \text{card}((\{i, j \mid i \in \text{NODES} \wedge j \in \text{dom}(G(i)) \wedge (G(i))(j) = \text{True_execute}(j)) \}) \geq 2 \times \text{card}((\text{NODES})) + 1 \)}
\end{align*}
\]

Listing 2: Invariants of M0.

\[
\begin{align*}
\text{Event \ Go\_faulty} & \quad \text{ANY NODE} \\
\text{Where} & \quad \text{\(\forall i \in \text{corr} \rightarrow \text{G}(i) \subseteq \text{Faulty} \)} \\
\text{Then} & \quad \text{\(\text{Pre_set} \subseteq \text{NODES} \)} \\
\text{End}
\end{align*}
\]

Listing 3: Event ‘Go_faulty’.

\[
\begin{align*}
\text{Event \ Primary\_change} & \quad \text{ANY NODE} \\
\text{Where} & \quad \text{\(\forall i \in \text{Pre_set} \rightarrow \text{Pre} \subseteq \text{NODES} \)} \\
\text{Then} & \quad \text{\(\text{Pre} \rightarrow \text{max}(\text{NODES}) \)} \\
\text{End}
\end{align*}
\]

Listing 4: Event ‘Primary_change’.

\[
\begin{align*}
\text{Event \ request} & \quad \text{ANY m} \\
\text{Where} & \quad \text{\(\forall i \in \text{NODES} \rightarrow m \notin \text{cache}(i) \)} \\
\text{Then} & \quad \text{\(\text{cache} \rightarrow \text{cache} + \{\text{pre} \rightarrow \text{cache}(\text{pre}) \cup \{m\} \)} \\
\text{End}
\end{align*}
\]

Listing 5: Event ‘request’.

\[
\begin{align*}
\text{Event \ commit} & \quad \text{ANY node} \\
\text{Where} & \quad \text{\(\forall i \in \text{NODES} \rightarrow \text{cache}(i) \)} \\
\text{Then} & \quad \text{\(\text{cache} \rightarrow \text{cache} + \{\text{node} \rightarrow \text{cache}(\text{node}) \cup \{m\} \)} \\
\text{End}
\end{align*}
\]

Listing 6: Event ‘Commit’.

5.1. The Initial Model. The purpose of the initial model is to define the success conditions for PBFT. This model should be as abstract as possible and specify the critical liveness and agreement properties to make them valid. We begin by defining the initial Event-B context as shown in Listing 1. In this context, we define a carrier set Message to represent
the different message formats introduced in Section 4. The carrier set Result represents the execution result produced by each node. It may contain many kinds of execution results in voting, calculation, election, etc., but they can all be regarded as elements of the set Result. To formalize the assumption A1, we define the constant NODES as the set of all network nodes and axiomatize that it is a finite and not empty subset of natural numbers. Besides, CORR is modeled as the set of all the correct initial nodes and it is a subset of NODES. To ensure that the initialization satisfies the premise of the PBFT, which bounds the number of faulty nodes, we add the axiom @axm5 (3 * card(CORR) ≥ 2 * card(NODES) + 1).

We define a relation True_execute from messages to results, which indicates the results that should be returned by a correct execution, and we consider the worst case, where the Byzantine node, noted as faulty nodes, return the special value FAULTY.

In our initial model, each node has a buffer where all the messages received and processed are stored. As shown in Listing 2, we formalize the final view of each node as G, which is a set of relations between each node to its result pairs. Since not all the messages contain their execution result, we define the partial function (Message→Result) from messages to results so that each node can save the messages being executed with their result in the buffer like [(M1, R1), (M2, R2) ...]. At termination, each node i has its view G(i). Besides, we define a variable Cache to serve as a buffer to store all the received messages for each node, which is a total function from nodes to messages. @inv5 specifies that the set of messages stored in G is a subset of that in Cache because some messages have been received but not yet executed.

The variable Pre contains the primary node and Pre_set is the set of all nodes which have served as a primary node in some rounds. The other important variables we have defined are corr and Faulty, the former is the set of all correct nodes and the latter contains all the faulty nodes. The union of these two sets constitutes the entire set NODES, and their intersection is the empty set. @inv8 specifies that all correct nodes will reply the correct execution results, @inv9 specifies the agreement property of the model. The agreement property emphasizes that the cardinality of the set of nodes replying with the correct execution results should exceed f + 1 (f being the number of faulty nodes), which can be specified by

\[ \text{card} \left( \forall i, j \in \text{NODES} : j \in \text{dom} (G(i)) \land (G(i))(j) = \text{True.execute(j)} \right) \geq 2 \times \text{card} (\text{NODES}) + 1. \]

Listing 7: Event 'Reply_s'.

Listing 8: Invariant of M0.

We define the event Go_faulty in Listing 3 to convert a correct node to a faulty one as below.

The first guard @grd1 stipulates that the parameter should be an element of the set of correct nodes, the guard @grd2 specifies that the number of correct nodes should be greater than (2 * card(NODES) + 1)/3 after removing the node from the set corr. The third guard @grd3 specifies that the node can only be faulty during the phases Commit and Reply. Besides, we define the event Primary_change in Listing 4 which should be triggered before each client sends a message to the primary and the primary node is faulty. As shown, each node will be elected as the primary node in turn from high to low by taking the maximum value, and each node has the same chance of being elected as the master node. We define a variable Pre_set as the set of all the nodes selected as primary node. When all nodes have taken turns to become primary nodes, the variable Pre_set will be reset to an empty set. The primary node will be reelected from all nodes, and the variable Pre_set will continue to grow as...
new primary nodes are selected. This formalizes requirement 3.

For the event Request in Listing 5, the parameter m is the new message which will be added to the buffer, and @grd2 guarantees that m has not been received before. @grd3 guarantees that the primary node must be correct. Then we use a relational override to update the buffer of the primary node in @act1.

As shown in Listing 6, the event Commit describes the process of all nodes receiving and confirming messages from the primary node. In particular, it is an asynchronous process. The parameter m is the message sent by the primary node to all other nodes, and node is any node except the primary node. This event ensures the agreement property by requiring that the execution results of at least f + 1 different nodes are identical.

As shown in Listing 7, we define the event Reply, which will fire when the agreement property above holds. When fired, the event updates the global variable G with the new consensus result and the execution result. It is also modelled as an asynchronous process. According to the validity of the execution results, there should be at least f + 1 identical result from different nodes. Therefore, we propose the theorem @inv17 in Listing 8, which ensures the agreement property.

In our first refinement, we start to model the details of the protocol. To more specifically refine the serial number and content of the message defined in M0, we define a carrier set Value to represent different operations sent by the client in the context, and a constant called contents, which is an injection from Message to a set of pairs. In the set of pairs, the first element is a member of natural numbers and the second element is a member of Value. The natural number records the sequence number of the message sent by the client to the primary node. Together with the value, it ensures the uniqueness of each message.
@inv7 specifies that it is a monotonically increasing function. Besides, we refine the Value type by two constants Correct_value and Faulty_value, which specify the condition where the faulty nodes forward the faulty value during the prepare and commit process (@inv6 and @inv8). @axm3 and @axm4 specify the relationship between the two constants. $H$ is the high level of the water marks interval and we set it to 2 for tests in Listing 9.

For the machine, in Listing 10 we define a natural number variable $n$ to represent the low level of the water mark interval, and a total function View from nodes to nodes, which represents the view of the primary node for each node.

As shown in Listing 11, the event Low_water is defined to change the variable $n$. $S$ is the set of all messages which are under consensus and have at least $f + 1$ identical execution result from correct nodes(@grd1-@grd5). Then we assign the sum of 1 and the maximum value of the sequence number of messages in $S$ to variable $n$.

Another relevant event we have defined is under_low_water. As shown in Listing 12, for any node in the network, if there is any message that has not executed whose sequence number is less than $n$, then these messages will be updated with the execution results from other correct nodes(@act1).

The event Request, Commit, and Reply are all refined with the specific data types of messages and tighter restrictions. For Request in Listing 13, we define two parameters $m1$ and $m2$. $m1$ is any message sent to the primary node by the client and $m2$ is the content of the message. @grd3 guarantees that the message sent by the client is different from the messages which have been received by any node. @grd4 and @grd7 guarantee that the client cannot send duplicate messages to the primary node. @grd5 specifies that the sequence number of the messages sent is monotonically increasing by 1. @grd6 formalize the assumption 6, and it indicates that the client should wait when there is one more message than $n$ in processing, therefore, this distributed algorithm can run stably without causing the system to crash due to weak synchrony. @act1 updates the cache of the primary node.

For the event Commit, we add the constraint of water mark interval by @grd1 and @act1 as shown in Listing 14 to update each node’s view of the primary node. It is similar for event Reply_s, we add the constraint of water mark interval.

5.2. The Second Refinement. The second refinement focuses on the consensus steps. We modelled the consensus phase by these five steps: Request, Pre-prepare, Prepare, Commit, and Reply. Request refines Request, Pre-prepare refines Commit, and Reply_s refines Reply. Besides, event Pre-prepare, Prepare_check, Commit_s, and Commit_check are defined to accomplish the whole process. These steps are all under weak synchrony corresponding to the real situation.

We introduce five new variables $G_p$, $G_pre$, $G_r$, $G_pre_check$, and $G_r_check$ with the invariants given by Listing 15. $G_p$ is a total function used to save the message information received during the request and pre-prepare process, which refines the variable cache in the previous machine, @inv6 specifies their relationship. $G_pre$ is the total function from nodes to another partial function which maps nodes to a set of message information. It serves as a cache during the prepare process. $G_pre_check$ is a total function from nodes to message information, which serves as a cache to store the received message under consensus. $G_r$ and $G_r_check$ are similar to $G_pre$ and $G_pre_check$, and they are used during the event Commit and event Commit_check.

The event Prepare of Listing 16 models the process where all nodes in NODES\{Pre\} send a message to nodes...
in NODES, any faulty node can forward false messages with the same sequence number, view, but the different message value (@grd7, @grd8, @grd9, @grd12, and @grd13). To make this process keep weak asynchronous, @grd6 is used to guarantee that all messages are sent with a sequence number that belongs to the water mark interval.

The event Prepare_check in Listing 17 is modelled to check the validity of the messages received during the event

Listing 15: Invariants of M2.

Listing 16: Event 'Prepare'.

Listing 17: Event 'Prepare_check'.

in NODES, any faulty node can forward false messages with the same sequence number, view, but the different message value (@grd7, @grd8, @grd9, @grd12, and @grd13). To make this process keep weak asynchronous, @grd6 is used to guarantee that all messages are sent with a sequence number that belongs to the water mark interval.

The event Prepare_check in Listing 17 is modelled to check the validity of the messages received during the event
Prepare: at least $f+1$ message received with the same sequence number should be identical. The message that passes the check is stored in variable $G_{\text{pre\_check}}$. It is also a weak synchronous process by the guard $@grd3$.

The events Commit and Commit_check have a similar logic, except for the different stored data structure. Therefore, we do not detail them here.

### 5.3. The Third Refinement

Based on the former three models, we have achieved the agreement property of the PBFT. Therefore, the third refinement focuses on the view change function, which is very important to ensure the liveness property. We refine the Faulty variable with two variables Crash and Evil. Crash is the set of nodes that can receive messages but cannot forward them. Evil is the set of nodes that will forward false values during Prepare and Commit, and reply the incorrect execution results to the client during the Reply phase. As shown in Listing 18, $@inv1$ specifies the relation among Faulty, Crash, and Evil. $@inv2$ and $@inv3$ specify the agreement property, that is to say, the cardinality of the corr should be greater than or equal to the cardinality of Crash and cardinality of Evil, which is consistent with the $@inv3 (3*\text{card (corr)} \geq 2*\text{card (NODES)} + 1)$ in the initial model. In this refined model, once the primary node crashes, there should be an event to change the primary node, and all other nodes should change their view so that the system will not enter infinite waiting. Therefore, we define a variable View_change, which is a total function from nodes to another partial function, which maps nodes to natural numbers.

We defined the event View_change in Listing 19. It takes a sender, a receiver, and the primary node as parameters, $@grd2$ specifies that the primary node crashes. $@act1$ specifies that any node in NODES sends the message which declares the next primary node to all other nodes in NODES. It is an asynchronous process. This event will be fired until all guards in event Primary_change are met.

As shown in Listing 20, the event Primary_change models that the next primary node receives at least $f + 1$ identical messages from different nodes, which indicates the next primary. We introduce a variable $S$, $@grd1$ specifies that $S$ is a subset of the domain of the following primary node’s View_change. $@grd2$ specifies that nodes in $S$ send the message which indicates the primary node max(NO-DES$(\text{Pre\_set})$, and we elect the master node in turn according to the node number($@act1$).

We define another two events Go_Evil and Go_Crash to refine the event Go_faulty. In particular, they specify that the primary node will not become evil, but can crash. And we refine the consensus process like Pre_Prepare with the guard $\text{Pre\_Crash}$, which means that if the primary node crashes, it cannot forward messages. Then the event Primary_change may fire. As for event Prepare, we add the guard send#Crash, which means that crashed nodes cannot send messages. In our system, we allow at most $f$ nodes to crash as specified in R2.

### 6. Proof Engineering

The proofs of the PBFT model have been mechanized thanks to the Rodin framework. The framework in here was used as a proof obligation generator and as an environment to discharge generated proofs through user interaction. The framework contains built-in solvers and is also connected to external SMT solvers. The basic machinery available within Rodin allows for the automatic generation of proof obligations for invariants, event convergence, refinements, and theorems. An invariant is true initially and preserved by each event. Event convergence is established by introducing a variant, an expression yielding a natural number or a finite set. Each convergent event must decrease the variant strictly. Event-B also provides anticipated events which do not increase the variant. We use these features to generate proof obligations ensuring the stabilization property of the PBFT.

The main properties of PBFT are agreement and liveness. The former states that regardless of the view, results provided by correct nodes for a given message should be the same. The assumption R1 has been proved in Section 4.

The liveness property holds under the hypotheses that messages sent by clients will finally be processed and be weak fairness [29] over events $E$ in a context $H$. Learning from the proof method proposed in [12], we propose proof obligations for establishing the stabilization of a given property $Q$:

1. $\land_{e\in E \setminus C}(H \land V = v \Rightarrow [e](V \leq v))$ (Generated by Rodin for anticipated events): anticipated events do not increase the variant
2. $\land_{e\in E}(H \land V = v \Rightarrow [e](V < v))$ (Generated by Rodin for convergent events): convergent events make the variant decrease
3. $H \land V \neq \emptyset \Rightarrow \bigvee_{e\in C\text{enabled}(e)}(\text{Manually added as a theorem to be proved): some of the convergent events are enables while the variant is not empty}
4. $H \land V = \emptyset \Rightarrow Q$(Manually added as a theorem to be proved): when the variant is empty, the targeted property is satisfied

$C \subseteq E$ is a set of convergent events, and $V$ is a set expression called variant. Given an event $e$, $[e](p)$ denotes the
enabled convergent events should eventually be fired for the variant to decrease, and anticipated events do not increase the variant. A variation of this proof rule may be used when \( Q \) is reached before the variant \( V \) becomes empty. Obligation (3) can be changed as follows:

\[
\exists j \in \text{dom}(G(i)) : (G(i)(j) = \text{True_execute}(jj)) \rightarrow 2 \times \text{card}(\text{NODES}) + 1
\]
In Event-B, the system should satisfy the termination property, which means there should be a suitable variant to guarantee the convergence of the system. Our model uses the constant function contents to store the execution results for every node. Initially, we define a set that contains all messages that are not yet agreed on and executed at a specific time. Then, it will decrease until it becomes an empty set.

Another important thing is that our model ignores the difference in computing performance of each node, which is reflected in the model by the fact that message transmission and execution will be complete in one instant. Therefore, we do not need to consider the situation that often occurs in actual development. For example, a correct node may be slow due to poor performance, and it is not straightforward in specifying a waiting time.

In Table 2, we give proof statistics of the development of the Rodin tool. These statistics measure the model’s size, the proof obligations generated and discharged by the Rodin platform, and those inter-actively proved. The table shows that 56% (123/219) of automatic proof is achieved. These results are acceptable since our formal model uses the card function to restrict the number of the relationship between nodes, which complicates the verification. Also, there are set comprehensions, disjunctions, and strict subsets, so many proof obligations cannot be proved automatically.

In this paper, we proposed a mechanized correctness proof of the PBFT, widely used in blockchain systems. We formally developed the PBFT model by horizontal refinement by implementing core mechanisms like primary node change, water mark interval, and parameterized message types. The result shows that if the proportion of Byzantine nodes is under 1/3, the system can reach an agreement finally, and it will not fall into a deadlock for each message. This work provides a reusable model for developing consensus mechanisms with detailed message types rather than just values. Refactoring the development to use simpler datatypes may lead to improved levels of automatic proof and therefore improve the potential for reuse. However, there are still some limitations. For example, suppose the consensus protocol of the consortium chain makes significant changes to PBFT. In that case, there is no guarantee that the changed parts can also meet the verification conclusions in this article. Moreover, the manual proof effort required by this work may be too high to be reused in more complex developments.

In future work, we intend to build adversary models of distributed systems to test the reliability of PBFT, these are Sybil attacks, DDoS, etc. Another line of work is generating consensus protocol code for target systems and developing automated support for stabilization proofs.

### Data Availability

This is because all functions except the repeated code are included in the lists, and tables show the results.

### Conflicts of Interest

The authors declare that they have no conflicts of interest.
Acknowledgments

This work was supported by the Project of Science and Technology Major Project of Yunnan Province (202103AN080001-001, 202002AA100007, 202002AD080003), Yunnan Key Laboratory of Blockchain Application Technology (202105AG070005), and State Key Laboratory of Software Development Environment (no. SKLSDE-2022ZX-11).

References