Formal Verification of Classical Paxos for Distributed Consistency Based on SMT

Tengfei Li
East China Normal University
Shanghai, 200062, China
Email: tengfeili2015@gmail.com

Jing Liu, Haiying Sun
East China Normal University
Shanghai, 200062, China
Email: jliu,hysun@sei.ecnu.edu.cn

***Abstract*—Distributed systems lead to distributed consistency problem because of node failure and unreliable network when transferring information through the network. In the paper, we present an approach for formal verification of the classical Paxos protocol for consensus consistency. The specification is written in TLA+, Lamport's Temporal Logic of Actions. We verify the protocol using a SMT(Satisfiability Modulo Theory)-based approach. Specifically, we describe the Paxos protocol and verify if it is satisfiable using Z3.**

I. Introduction

When processing capacity of single node can not meet the need of the growing computing and storage tasks, and hardware upgrade (memory, disk, and CPU) leads to too expensive to pay, we need to consider the distributed system. However, distributed system is a topological structure with multiple nodes through network communication, in order to solve these problems, more mechanisms and protocols will be introduced. Further, this brings more challenges of distributed systems mainly including node failure and unreliable network when transferring information through the network. Those challenges leads to distributed consistency problem.

In order to solve distributed consistency problem, Lamport [3] proposed Paxos algorithm, which is a consistency algorithm based on message transfer model with highly fault-tolerance. Paxos has attracted the mind of many researches in the field of computer science, and it has been proved its effectiveness in distributed systems. However, there left a critical question how to guarantee its correctness of the algorithm. Charron-Bost [8] proposes a formal verification of a non-trivial Consensus algorithm using the proof assistant Isabelle/HOL. Rahli [7] specifies the Multi-Paxos protocol in EventML and proved its safety properties to be correct in Nuprl. Marzullo [9] presents a simple proof for Paxos and fast Paxos through the definition of safety and liveness. Delzanno [10] verifies distributed broadcast protocols bases on graph rewriting and graph transformation systems. Tsuchiya [11] proposes a bound model checking to check Pasox LastVoting algorithm through reducing state space of the distributed system. Kellomaki [12] provides the specification and proof of the consensus protocol of Paxos using PVS.

Those approaches focus on correctness of verifying Paxos algorithm. However, there is not enough to solve the Paxos and return the value that makes the algorithm satisfactory. In this paper, we present the specification and verification of Paxos sing a SMT(Satisfiability Modulo Theory)-based approach. Specifically, we describe the Paxos algorithm and check if it is satisfiable using Z3 constraint solver. Although there are several variant algorithms for Paxos, like, Cheap Paxos [13], Fast Paxos [14] and Vertical Paxos [15], we just verify a classical one (Basic Paxos), because during every round, each process first sends messages and then receives messages from other processes, and finally makes a local state transition. In the following of the paper, we adopt Paxos to represent Basic Paxos. So, our contribution is specification and verification of Paxos using Z3 solver.

Next section, we will present the preliminary of the algorithm Paxos and the SMT solver Z3. In section 3, we will model the distributed protocol with Z3 from the aspect of the process of communication. The verification of the model and the discussion of verification result will be present in section 4. In last section, we will conclude the work.

II. Preliminary

A. Paxos

Paxos is a classical protocol for distributed consistency, proposed by Lamport [3]. Paxos protocol employs three classes of agents: client, proposer, acceptor and learner. The client distributes the number of proposal to every proposer. Then the proposer proposes to the acceptors to make decisions. The acceptors have the right to decide whether accepts the proposal. An acceptor will promise to the proposer from which the proposal is the max value among all the accepted proposal. If one of the acceptors accepts preliminarily the proposal, the proposer will send the message to ascertain whether the proposal has been accepted by the acceptor. After that the acceptor responds the corresponding message to the proposer. Meanwhile, the acceptors send the number of proposal to the learner. Otherwise, the acceptor reject the proposal immediately. Learner learns from the acceptors by the proposal, and distinguishes the final result through the decision of the majority of the acceptors. The specific progress is shown in Figure I:

The process of communication can be simplified to the following progress:

- *Prepare*: The proposers prepare the proposal and send the proposal with *Prepare* message to all the acceptors.

- *Promise*: The acceptor responds to the proposer if the *Prepare* message is accepted. Otherwise, the acceptor sends nothing back to the proposer. Specifically, if the number of the proposal is great than that of the proposal has been accepted before, and then the acceptor sends a *Promise* message back to the proposer In this way, the acceptor promises that it will not participate in any round smaller than i and it will stick to this promise. Along with the promise, the acceptor sends the last value it has voted and the associated round.
- *Accept*: After collecting a quorum of n promises for round i from the acceptors, the proposer sends an *Accept* message to all the acceptors asking to vote for a value selected as follows: A value v proposed by the proposer, if no acceptor in the quorum has ever voted; The value vval in the promises that is associated with the highest round, otherwise (note that there can exists at most one such value, since the only value that can be voted in a round is the one proposed by the proposer that is responsible of that round).
- *Learn*: If an acceptor receives an *Accept* message, and if it has not promised otherwise, it votes for the value in the message and sends a *Learn* message to all the learners to let them know about the vote. Acceptors votes only once in each round.

B. Z3

Z3 is developed by L. de Moura and N. Bjorner at Microsoft Research[1], as a popular SMT solver owing to its efficiency to verify the satisfaction of the system.

It describes a system with the syntax of first-order logic, arithmetic, bit-vectors, arrays and tuples. Each category of the syntax need to be declared with the command of *declare-const* and *declare-fun*. And then we will declare *assert* to describe the formula. The command *check-sat* determines whether the current formulas are satisfiable or no. The expressiveness and corresponding operators are shown in Table I:

TABLE I
SYNTAX OF Z3

Expressiveness	Operators
Propositional Logic	and, or
Arithmetic	Addition, Subtract, Multiplication, Division, Integer Division, Modulo and Remainder operators
Bit-vectors	bvadd, bvsub, bvneg, bvmul, bvor, bvand, bvnot, bvule, bvult,...
Arrays	select, store
Datatypes	Record, Scalar, Recursive datatypes
Quantifiers	forall, exists

III. MODELING OF PAXOS

Poxos provides an approach to deal with distributed consistency. Among its four actors, proposer and acceptor play a key role in the algorithm. The algorithm give a constraint condition of safety requirement:

- Only a value that has been proposed may be chosen.
- Only a single value is chosen.
- A proposer never learns that a value has been chosen unless it actually has been chosen.

This safety requirement for consistency is described as:

$$Consistency \triangleq \forall\ v_1, v_2 \in \mathcal{V} : \phi(v_1) \wedge \phi(v_2) \Rightarrow v_1 = v_2$$

, where $\phi(v)$ means that v has been chosen.

The safety requirement ensures the uniqueness of the chosen values. That is, consistency is guaranteed in distributed environment because of uniqueness even if the information is tampered illegally.

How the unique value is chosen depending on the predicate $\phi(v)$. From this definition, it is obvious how a process learns that a value has been chosen from messages $send('2b', v, b, a)$. Predicate ϕ is defined as follows:

$$\phi(v) \triangleq \exists\ b \in \mathcal{B}, \exists Q \in \mathcal{Q}, \forall a \in Q, send('2b', v, b, a)$$

where $\mathcal{B}$ is a set of ballots. $\mathcal{Q}$ is a subset of all the acceptors, and Q is a subset of $\mathcal{Q}$.

Paxos algorithm is divided into two phases. In the first phase, the proposer sends the message to a majority of acceptor and receives a responds from the acceptors whose proposal is highest-numbered ballot. In phase 2, the proposal sends a *accept* request to those acceptors with a value v, and the acceptors accept the proposal.

A. Phase 1

In the first stage, the proposer and acceptors finishes the first handshaking for communicating each other. The proposer sends the *prepare* request to the acceptors, and gets the *promise* from the acceptors after their judgement.

TABLE II
PHASE 1 OF PAXOS

Phase 1a	A leader selects a ballot number b and sends a $1a$ message with ballot b to a majority of acceptors. It can do this only if it has not already sent a $1a$ message for ballot b
Phase 1b	If an acceptor receives a $1a$ message with ballot b greater than that of any $1a$ message to which it has already responded, then it responds to the request with a promise not to accept any more proposals or ballots numbered less than b and with the highest-numbered ballot (if any) for which it has voted for a value and the value it voted for in that ballot. That promise is made in a $1b$ message.

[1]https://github.com/Z3Prover/z3

a) Phase 1a: Phase 1a applies the command $prepare(type, bal)$ to communicate between the proposer and acceptors. Listing 1 presents the model of phase 1a. First, the sort type $Message$ is declared to represent the type of the transmitted message ($OneA$, $OneB$, $TwoA$, $TwoB$) and a set of message $msgs$. In order to express the highest-number ballot acceptor and the highest ballot, $maxBal$, $maxVBal$, $maxVal$ are adopted when the acceptors decide which one proposal should be sent or whether is essential to send a proposal to the proposer. The sort $Acceptor$ is a set of acceptors and an acceptor a is declared as the type of $Acceptor$. Also, bal represent a ballot as the type of Int.

Listing 1. Model of a Phase 1a

```
(declare-sort Message)
(declare-const OneA Message)
(declare-const OneB Message)
(declare-const TwoA Message)
(declare-const TwoB Message)
(declare-const msgs Message)
(declare-const maxVBal Int)
(declare-const maxBal Int)
(declare-const maxVal Int)
(declare-sort Acceptor)
(declare-const a Acceptor)
(declare-const b Int)
(declare-const type Message)
(declare-const bal Int)
(declare-fun prepare (Message Int) Bool)
(assert (= type OneA))
(assert (= bal b))
(assert (= (prepare type b) true))
(check-sat)
```

A function $prepare$ is declared for proposer to send the type $OneA$ and the ballot b to acceptors. Z3 checks the proposition with the command $check-sat$.

b) Phase 1b: This subphase employs command $promise(type\ bal\ maxVBal\ maxVal\ acc)$ to describe that the acceptors send the $promise$ to the proposer after determining the proposal to be sent.

Listing 2. Model of Phase 1b

```
(declare-const acc Acceptor)
(assert (= type OneA))
(assert (> bal maxBal))
(declare-fun promise (Message Int Int Int
                      Acceptor) Bool)
(assert (= (promise type bal maxVBal maxVal
                      acc) true))
(check-sat)
```

As shown in Listing 2, a constant acc with $Acceptor$ type is declared to represent the acceptor who needs to send the proposal to the proposer. It is worth noting that the sent ballot bal is the highest ballot.

B. Phase 2

The second phase mainly send the ballot and value of the highest-numbered proposal among the responses by the proposer to the acceptors. After that, the acceptors vote for the message's value v in ballot b. The description is list as Table. III:

TABLE III
PHASE 2 OF PAXOS

Phase 2a	If the leader receives a response to its $1b$ message (for ballot b) from a quorum of acceptors, then it sends a $2a$ message to all acceptors for a proposal in ballot b with a value v, where v is the value of the highest-numbered proposal among the responses, or is any value if the responses reported no proposals. The leader can send only one $2a$ message for any ballot.
Phase 2b	If an acceptor receives a $2a$ message for a ballot numbered b, it votes for the message's value in ballot b unless it has already responded to a $1a$ request for a ballot number greater than or equal to b.

a) Phase 2a: Phase 2a presents that the proposer sends ballot b and the corresponding value v to the acceptors. Listing 3 declares a value v and c with type of Int. Z3 isn't enough to express the return of two values, so we build two function f and g to return the type of message and ballot b respectively.

Listing 3. Model of Phase 2a

```
(declare-const v Int)
(declare-const c Int)
(declare-fun f (Acceptor Message) Message)
(declare-fun g (Acceptor Int) Int)
(assert (= type TwoA))
(assert (forall ((a Acceptor))
                (and (= (f a type) TwoA)
                     (= (g a bal) b)
)))
(assert (exists ((v Int))
           (exists ((m Message))
                (and (= (f a type) OneB)
                     (= (g a bal) b)
                ))
))
(assert (forall ((a Acceptor))
           (exists ((m Message))
                     (= acc a)
)))
(assert (or (exists ((c Int))
                (and (> c 0)
                     (< c (- b 1))
           ))
           (forall ((m Message))
              (<= maxVBal c)
           )
           (exists ((m Message))
                (and (= maxVBal c)
                     (= maxVBal v)
           ))
))
(declare-fun accept (Message Int Int) Bool)
(assert (= (accept TwoA b v) true))
(check-sat)
```

After declaring the set of formulas, we declare a function $accept$ to send the type of message $TwoA$, the ballot b and the responding value v to the acceptors who send the $promise$ information. The $accept$ function needs to be checked satisfiable under the condition of the declaration of the former *assert*.

b) Phase 2b: Phase 2b guarantees that an acceptor votes for the message's value in the received ballot b. In the round, a message with the ballot appears. Listing 4 shows the model.

Listing 4. Model of Phase 2b
```
(declare-const val Int)
(declare-fun learn (Message Int Int
                   Acceptor) Bool)
(assert (exists ((m Message))
               (and (= type TwoA)
                       (>= bal maxBal)
               )
))
(assert (= (learn TwoB bal val a) true))
(check-sat)
(get-model)
```

IV. Formal Verification of Paxos

A. Verification result

The advantage of the Z3 solver is that the solution it returns is the boundary value, and we can know how to get the value of the system in the boundary of the formula constraints. After specifying the mode of Paxos using constraint condition, we get the solution that Z3 returns. The specific result is shown in the website[2].

The command *get-model* retrieves an interpretation that makes all the constraint true. The interpretation includes a solution of the verification result.

B. Discussion

We model Paxos protocol using Z3. But, there is some short of comparison with other related work.

V. Conclusion

In this paper, we present an approach to specify and verify Paxos protocol for distributed consistency.

References

[1] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.

[2] C. Barrett, P. Fontaine, and C. Tinelli. *The SMT-LIB Standard Version 2.5 Reference Manual*, Jun. 2015.

[3] L. Lamport. *Paxos made simple*. SIGACT News (Distributed Computing Column) 32(4), 51-58 (2001)

[4] Chand S, Liu Y A, Stoller S D. *Formal Verification of Multi-Paxos for Distributed Consensus*. International Symposium on Formal Methods. Springer International Publishing, 2016:119-136.

[5] Apasuthirat, Thanisorn. *A survey of formal verification of Paxos and a case study with an algebraic specification language* 2014

[6] Küfner P, Nestmann U, Rickmann C. *Formal Verification of Distributed Algorithms*. IFIP International Conference on Theoretical Computer Science. Springer, Berlin, Heidelberg, 2012:209-224.

[7] Rahli V, Guaspari D, Bickford M, et al. *Formal Specification, Verification, and Implementation of Fault-Tolerant Systems using EventML*. Nuprl Org, 2015.

[8] Charron-Bost B, Merz S. *Formal Verification of a Consensus Algorithm in the Heard-Of Model*. Intl.j.software & Informatics, 2009, 3:273-303.

[9] Marzullo, K., Mei, A., Meling, H.: *A simpler proof for paxos and fast paxos*. Course Notes (2013)

[10] Delzanno G. *Parameterized Verification and Model Checking for Distributed Broadcast Protocols*. Graph Transformation. Springer International Publishing, 2014:1-16.

[11] Tsuchiya T, Schiper A. *Using Bounded Model Checking to Verify Consensus Algorithms*. International Symposium on Distributed Computing. Springer-Verlag, 2008:466-480.

[12] Kellomaki P. *An Annotated Specification of the Consensus Protocol of Paxos Using Superposition in PVS*. Tampereen Teknillinen Yliopisto, 2004.

[13] Lamport L B, Massa M T. *Cheap paxos*. IEEE, US 7249280 B2[P]. 2007.

[14] L. Lamport. *Fast paxos*. Distributed Computing, 19(2):79?103, 2006.

[15] Lamport L, Malkhi D, Zhou L. *Vertical paxos and primary-backup replication*. ACM Symposium on Principles of Distributed Computing. ACM, 2009:312-313.

[2]https://github.com/EcnuTengfei/paxos-z3